UNIVERSITY OF OSLO
Department of Informatics

Generic Software Distribution and Deployment

Master thesis

Arve Knudsen

31st January 2006

Acknowledgments

My family and my supervisors Hans Petter Langtangen and Halvard Moe, thank you for keeping your faith in me. A special thanks to Professor Langtangen for providing constant encouragement and invaluable input in this difficult process.

Contents

List of Figures

List of Tables

Chapter 1. Introduction

Distribution and deployment of software is a recurring theme in dealing with computers, whether the emphasis is on the former or the latter depends on the individual's level of involvement. As a user one will typically deal with deployment tasks on a fairly regular basis, in order to add to one's system or as part of maintaining existing installations. If one is to install something or apply an update, acquiring the software generally involves some contact with distribution mechanisms. As a provider (developer) of software on the other hand, a deeper level of involvement with the distribution process is required, since it governs how one's products are offered to users.

PyPkg is intended as an integrated, generic, system for handling tasks related to distribution and deployment of software, in a way that serves both the user and provider perspective. The two are integrated since they are ultimately facets to the same higher purpose: to deliver software into the hands of users so it can be applied. PyPkg approaches the problem by presenting facilities for distribution and deployment, aimed at providers and users respectively. These facilities together form a system, through which users can obtain and deploy software from providers in one seamless operation.

The distribution facilities serve to establish a layer of software providers, that describe themselves and their products in a generic language (expressed through XML). The deployment facilities, which collaborate with the distribution layer, manifest themselves as Python [36]-based tools that follow a common, platform-agnostic, scheme for deploying software on target platforms. As of the current version of PyPkg, deployment tasks are served by a single tool, a fully automated installation client. Also, PyPkg tools are at this stage invariably bound to the Linux platform.

The focus on Linux during development was a conscious decision, a way to keep complexity down. This platform is somewhat of a moving target, given the differences between individual Linux distributions, so it presents a significant challenge in itself. As such, this thesis is largely coloured by a Linux-centric perspective.

We will now go on to present a concrete scenario involving distribution and deployment of a software suite for simulating tsunamis. This example should give a suitable impression of the complexity involved in providing Linux users with a less than trivial software package, and, ultimately, how PyPkg may be of help in dealing with this complexity.

1.1. A specific problem scenario

Scientists at the Scientific Computing department of the Simula Research Laboratory [43] wish to distribute a base set of tools for performing tsunami simulations, the *Simula Suite*. The suite is originally being composed in support of projects conducted at the Norwegian Geophysical Institute, but Simula would like to distribute it freely for the benefit of fellow scientists. Inspired by the Python Enthought Edition ("Enthon") [17] software bundle for Windows, it con-

tains for the most part Python programs and components. Numerical engines in C/C++/Fortran underlying Python code are to ensure good performance.

The suite is a mix of programs and components for software development, that are developed independently and provided under a license supportive of free distribution. Given that the software is originally provided by many independent projects with little coherence in terms of deployment, the suite's constituent parts would be unwieldy to install and subsequently manage (update, uninstall) by themselves. The Python-based products are for the most part relatively easy to install, since they almost exclusively use the Distutils [12] framework which is standard with Python. The rest of the suite's components present a variety of installation methods however, many must be compiled from source code (C/C++/Fortran), Python itself for instance, which can be an error-prone and time consuming process. The products constituting the Simula Suite are presented in Table 1.1.

As evidenced by the table of Simula Suite components, the suite is quite complex in nature. As such, some consideration should be put into its composition on the provider's (Simula's) part. That is to say, one should look into ways of streamlining the deployment process on targeted platforms. On Windows, one may simply look to Python Enthought Edition for guidance. Enthought takes the approach of bundling everything into a single Windows Installer [28] package, which is a simplistic but decidedly practical approach. It means the provider can distribute a single self-contained executable that will install everything on the user's machine so that it will work out of the box. The whole bundle can also be uninstalled as a single entity.

On the other hand, there's no such obvious solution for Linux at least. All components of the suite support deploying on Linux through their individual deployment procedures, but there's no clear alternative for merging them all into a single deployable representation. Unlike the standard integrated Windows Installer service found on Windows, integrated deployment support on Linux tends to be tied to individual distribution architectures (Red Hat, Slackware, Debian etc.). Therefore the "universal" solution for distributing third party software to Linux users has typically been as platform-independent source code archives (which is true for the software included in the Simula Suite). Due to the complexity of deploying on this platform, the remainder of this section is devoted to investigating possible solutions.

Another concern especially pertinent to Linux is that of user rights; Linux systems are normally quite restrictive in terms of how a normal user may modify his/her surroundings, but provide no real facilities for installing software outside of system-wide locations. For Simula it is imperative that users without administrative rights be able to make use of the Simula Suite, so the chosen deployment solution must reflect this.

1.1.1. Manual deployment

Installing software using source level procedures (e.g., Distutils or Make [20] deployment scripts) can be a tedious and picky process, for some packages more than others. In addition to sheer difficulty of building and/or installing, some packages also present rather complex requirements on the runtime environment. That is, they typically depend on other third party software to be present in the host system. For instance, the Python interface to VTK (part of the Simula Suite) requires that the `Tkinter` Python module be present and functional, VTK and `Tkinter` must also use the exact same version of the Tk library.

One particular Simula Suite component sticks out like a sore thumb in this respect: SciPy

Name	Deployment method	Description
Python	GNU Autotools	Python interpreter/runtime
PyQt	SIP	Python wrapper for Qt
PMW	Distutils	Python Mega Widgets
MayaVi	Distutils	Scientific data visualizer
ParaView	CMake	Parallel visualization application
Triangle	Make, manual	Mesh generator
IPython	Distutils	Improved Python shell
Gnuplot	GNU Autotools	Plotting utility
Gnuplot.py	Distutils	Python interface to Gnuplot
Epydoc	Make	Documentation tool for Python
Matplotlib	Distutils	Python plotting library
F2PY	Distutils	Fortran->Python interface generator
PIL	Distutils	Python Imaging Library
Psyco	Distutils	JIT compiler for Python
SciPy	SciPy Distutils	Scientific tools for Python
epsmerge	Manual	Merge EPS files
ScientificPython	Distutils	Python modules for scientific computing
PySparse	Distutils	Sparse matrix computations in Python
R	GNU Autotools	Language for statistical computing
RPy	Distutils	Python interface to R

Table 1.1. The Simula Suite

[24]. It is a Python library, but parts of it are implemented in compiled languages (C, C++, Fortran). That parts of the library must be compiled doesn't present an immediate problem to the user, since it uses the standard Distutils procedure for building and installing ("python setup.py install") but the user can get exposed to some "interesting" errors during compilation. For instance, it is not entirely unusual to see C++ code (not SciPy specifically) fail to compile as new (and increasingly strict) versions of the GNU C++ compiler (the customary C++ compiler on Linux) are released. One may also run into binary incompatibilities: if the Fortran parts of SciPy are compiled with a different symbol naming convention (one versus two added underscores) than Fortran libraries it is linked with, mayhem will ensue.

The most problematic aspect of SciPy though is its external dependencies. It directly depends on Python (obviously) >= 2.1.x, Numeric [42] >= 21.0, F2PY [35] and BLAS/LAPACK. For the BLAS/LAPACK dependency one will normally want to utilize the ATLAS (Automatically Tuned Linear Algebra Software) [2] implementation, which automatically tunes the built library to perform optimally on the user's machine. According to SciPy documentation, ATLAS

may improve performance by a factor of 5 to 15 compared to the reference implementation. The price for this performance however is a highly complex build process, that requires the user to answer a series of questions. Once the build commences, it will be a lengthy process (several hours on a Pentium 4).

The built ATLAS libraries can't be installed directly either, according to the SciPy guide on this subject they must be combined with the reference BLAS/LAPACK libraries (a procedure not for the faint of heart) [41]. The finished ATLAS libraries must then be placed in the path of the system's linker, so they are properly included when building dependent programs (SciPy).

Given the error-prone and time consuming nature of deploying Simula Suite components from their source representation, it is something best not exposed to end users. Therefore we shall look into different ways of packaging the suite for deployment on Linux.

1.1.2. Deployment solutions at our disposal

Traditionally, when preparing a piece of software for (more or less) streamlined deployment on Linux, one would consider various Linux distribution-dependent package management systems. There are several popular alternatives, prominent examples include Red Hat RPM [3], Debian APT [48] and Gentoo Portage [44], ruling out one would affect the level of support for users of distributions employing the scheme in question. If we were to follow the traditional package management route when distributing the Simula Suite, we would likely find ourselves packaging each release in several different formats in order to meet users' needs. At least it would be much preferable from the user perspective compared to manual deployment. Some systems, notably APT and Portage, support automatic updates of installed software as well. For this to work, however, packages must be published through corresponding repositories.

A significant concern regarding typical package management systems is that they are not designed with user-specific installations in mind. Instead they are aimed squarely at installing software in system locations, and thus require the appropriate privileges. Many users wishing to install software lack administrative rights, or they don't see the need to clutter the system unnecessarily. In such cases conventional package management is of little help.

New designs are surfacing, though, that aim to provide package management-like facilities but free from ties to specific Linux distributions. Two initiatives that have come to our attention, in this respect, are Autopackage [23] and Zero Install [26].

1.1.2.1. Autopackage

Autopackage is not a package management system per se; rather, it presents a shell script API for writing binary[1] installation scripts that are portable across Linux. It does include facilities for adorning such scripts with automatic dependency handling though, and also offers management of installations. By management we mean that the API can record modifications of the filesystem during installation, in a database, so they may be undone when uninstalling. A special tool is provided for uninstalling software in the database.

Autopackage supports our requirement of being able to install without administrative rights, even completing an installation by configuring the user's environment. That is, any

[1] Autopackage scripts install from binary packages.

installed executables for instance will be added to the `PATH` variable of the user's shell.

The distribution model of Autopackage is rather simple. There are no special package repositories, instead packages are intended to be offered directly by the provider like any other type of resource. This would be beneficial for distributing the Simula Suite, as we wouldn't need to concern ourselves with any centralized distribution channel.

The most problematic aspect of Autopackage in our view is the script API approach. Since the installation management logic is embedded in supportive functions (part of the API), a script can completely bypass the management logic and modify the filesystem directly (intentionally or not). The same goes for uninstallation which is also delegated to the individual script. Autopackage does represent a possible solution for deploying the Simula Suite on end users' systems, but autonomous deployment scripts represent a step backward compared to the declarative package management approach. The PyPkg system is an example of the latter, where software packages are treated as simple data and the deployment logic itself is embedded within the client.

1.1.2.2. Zero Install

Zero Install represents a more sane approach to software deployment in our opinion, since it operates with declarative packages (primitive archives in fact) rather than autonomous scripts, but still doesn't quite fit the bill. The problem is that it is aimed at installing *applications*, i.e., it handles everything needed to run a program even if it isn't present in the user's system. When discussing Zero Install in this context, we are referring to the *Injector* implementation. The Zero Install Injector can run a program identified by a valid Zero Install resource identifier (a URI); it takes care of everything needed to run the program, acquiring packages for it and the whole dependency chain, installing them beneath the Zero Install cache and constructing a suitable runtime environment before the program is run. Programs can be removed simply by deleting the corresponding directory in the Zero Install cache.

The problem when considering Zero Install for deploying the Simula Suite is that it's not of much help for those parts of the suite that aren't complete applications. The suite also contains many modules for software development, these must be explicitly installed within the filesystem so they are available for development purposes. For instance, we could create a Zero Install "interface" (an XML document) for IPython, which is part of the Simula Suite, and place it at http://www.simula.no/suite/IPython. A user could then run the packaged application by executing the command "0launch http://www.simula.no/suite/IPython". This won't work for Psyco, also part of the Simula Suite, since it is not a runnable program. Rather, Psyco must be installed where it can be found by the Python interpreter.

1.1.3. Applying PyPkg to the problem

The PyPkg system, which is under development independently of the Simula Suite effort, offers a balanced view of the distribution/deployment problem. It is meant to reflect the needs of both providers and users, so the model includes support for the whole chain from initial distribution of software to its deployment on the systems of end users. The distribution facilities are even detached from the deployment side of things, so once software has been registered with the PyPkg distribution network it can be exploited for other purposes as well (presentation via HTML for instance). When it comes to deployment PyPkg shares some desirable qualities with Autopackage and Zero Install; it is free of ties to any particular Linux distribution and

supports user-specific installations, in addition to typical package management functionality like automated uninstallations (only partially implemented at this time, granted).

Autopackage and PyPkg stand out as the best suited alternatives for deploying the Simula Suite on Linux, given their cross-distribution support and that they both enable per-user installations. We've already mentioned our disapproval of Autopackage's approach of autonomous deployment scripts however. PyPkg represents the sane approach in this respect, from our point of view. A PyPkg package contains only files and information on how to integrate them with the host system, therefore the client exerts full control over its effect on the user's system.

The actual distribution of the Simula Suite through the PyPkg network will require some initial effort. Given PyPkg's project-oriented type of software distribution (see Section 3.2 for an explanation of the distribution model), an RDF project description must be created for each component of the Simula Suite according to the DOAP/PyPkg specification. Also, project descriptions must declare releases corresponding to component versions and associate them with PyPkg packages (a dedicated tool exists for this purpose). Lastly, each project must be registered with the central PyPkg project portal since clients access project information through this. The latter requirement isn't much of a problem given that the portal is controlled by Simula.

The creation of PyPkg packages for each part of the Simula Suite is likely to be the most difficult/time consuming aspect of the distribution process, given the time involved (especially with large packages like VTK) and likelihood of encountering errors in adapting the many source packages to PyPkg equivalents. Thankfully the PyPkg package authoring tool is quite usable already and many packages will build directly with built-in build rules, but some have their quirks and require special consideration. For instance, SciPy uses a special version of Distutils that would not work when invoked in discrete steps, and so required adaptation of the Distutils build rule. Section 4.1.2 shows in detail how a package is created for SciPy.

1.2. Implementation notes

All PyPkg programs, an installation client and a package authoring tool, are written in pure Python and feature a PyQt [39] GUI. The programs are heavily based on the common PyPkg framework, which is implemented as a Python package, `pypkg.base`. To improve performance, PyPkg programs use the Psyco [37] JIT compiler for Python when available. It should also be noted that our code relies on features introduced with Python 2.4, both on the language and library level.

Software packages in the PyPkg format are currently built for the Intel Pentium Pro processor architecture [53] only, which is supported by all newer (superseding Pentium Pro) x86 architecture processors (including the AMD Athlon range, Pentium 4 etc.).

The Python source code and documentation (generated with Epydoc) for PyPkg can be downloaded as a bzip2-compressed tar archive from http://home.broadpark.no/~aknuds-1/pypkg.tar.bz2.

1.3. Thesis structure

In writing this thesis we've employed a certain logical structure. The idea is that we first present a vision for PyPkg, stating what the system is to develop into, before we describe

the model for the current version of PyPkg, which is a step toward realizing the vision. This discussion is then complemented by a chapter on practical usage of PyPkg facilities, centered around the distribution and deployment of a certain software product. Finally, we conclude the thesis with a chapter devoted to reflection on PyPkg's current and future state.

Chapter 2. Vision

PyPkg is to provide a generic system for distribution and deployment of software, that much is clear, but it is too weak a notion to be of any real use in our work. The background is that we have a certain idea of how we think that software distribution and deployment should be conducted in general; this underlying intention needs to be refined in concrete terms, such that we make a commitment to how PyPkg is meant to manifest itself to the intended audience. In this regard it helps to keep in mind that our process should result in a tangible product, a software system to be exact. In order to ensure that the composed system, likely of significant complexity, corresponds to the expected result we establish an authoritative reference, that development must adhere to. This is the PyPkg *vision*, given that it describes the envisioned generic distribution/deployment system.

But let's not get ahead of ourselves. Before we present how it is we envision PyPkg, it makes sense to discuss how software distribution and deployment is in the general case conducted with existing means. This is to build an understanding of the complex problem PyPkg is meant to solve, and should as such form a suitable backdrop for the discussion of the vision itself.

2.1. The current situation

With PyPkg we wish to ultimately address how software is distributed and deployed in the *general* case, so we will concentrate our discussion in this section on how these processes are generally handled today. Highly advanced systems have been developed for these purposes, but they don't apply to the general case (those we are aware of at least). As an example the more advanced Linux package management systems tend to make deployment very convenient for users, but are tied to a certain platform (normally a specific Linux flavour at that). We consider instead the generally applicable method for getting software from developer to user, so that it can be put to use (i.e. fully installed), where distribution and deployment are disjoint processes. Concretely we're thinking of how software is distributed from individual websites to users, who proceed to deploy the software according to various procedures.

2.1.1. Distribution

Since the World Wide Web offers a generic[1] method for distributing software, as downloadable resources, it has effectively become a catch-all for this purpose. Even on Linux where there exist package management systems that are specially tailored for presenting/bringing software to users, developers themselves tend to resort to distributing through their Web pages since there is no single agreed-upon distribution channel for such systems. The problem with the Web being the least common denominator is that it doesn't provide special-purpose facilities (not a problem with the Web per se). Software is but one kind of digital content available from websites, and general tools for browsing and searching the Web provide little assistance in locating content of a

[1] In the sense that it's not particular to any platform or purpose.

specific kind. Additionally, the sheer amount of software, coupled with incoherent presentation, can itself prove overwhelming.

2.1.2. Deployment

There are not only difficulties involved in getting hold of software, deploying it on target systems can present even more of a challenge. The exact procedure depends essentially on the individual software provider. Different platforms vary in their support for deployment, some are developed to a greater degree than others in this regard.

On Windows support for software installation and uninstallation is integrated through the Windows Installer service since Windows 2000 [28]. As a result software for the Windows platform is normally installed through a simple Windows Installer-compatible graphical "wizard", and eventually uninstalled via an integrated Windows tool.

On Linux things are not so rosy, there being no standard deployment service à la Windows Installer. Instead users are in the general case stuck with generic archives, which typically contain some sort of installation script along with documentation on its intended use. Sometimes these installation scripts (Make scripts for the most part) support subsequent uninstallation, but in a generally unsafe manner (blindly removing files corresponding to what was/would be installed).

2.1.2.1. Dependencies

Also, Linux programs tend to (more often than not) require certain other programs to be installed beforehand. Such dependencies tend to come about as a side effect of the Free Software [21] development model, where projects are prone to depend on other software distributed under a Free license (e.g. GPL [19]) rather than "reinventing the wheel". Dependencies on third party software can of course occur in conjunction with Windows software too, but in practice far less often. The convention with Windows Installer packages seems to be to include necessary components with the installation instead, an approach that will at least enlarge the installation but saves the user hassle.

2.1.2.2. Updates

Another deployment related problem is that of updating software once installed. This is particularly intertwined with the distribution process, since one must first be aware that an update has been released and then acquire it, before it can be applied. Considering how often updates are typically issued these days, this process can quickly become a regular ritual.

Apart from receiving updates, deploying them can be a more or less daunting experience depending on the method employed. Installation scripts of the Windows Installer type tend to handle the transition from old to new version automatically, indeed the Windows Installer service has built-in support for 'upgrading' and 'patching'[1] installations [28]. Generic Linux installation scripts on the other hand can not be trusted to have this functionality, in the general case they will overwrite the files of an old version but not remove any remnants.

[1] Patching as opposed to upgrading presumably meaning that the installation is modified in place rather than replaced.

Convenience on the user's part is but one concern with regard to updating installations. Security is becoming more and more of an issue in the digital environment, with constant reports of vulnerabilities in programs. The same technology that enables developers to transmit software fixes in rapid succession (the Internet), lets knowledge of security breaches spread among subversive elements. Given the short timeframe involved in such situations it is key that updates are deployed with the highest possible efficiency, which can not be said of the current disjoint model where distribution and deployment are independent processes. Maximum efficiency in this respect can only be achieved with some integration between the two.

2.1.3. Software directories

Software directories of the Freshmeat [33]/Sourceforge [34] variety alleviate the problem of software distribution via the Web, by aggregating information about software projects and presenting them and their products in a consistent manner to users. Finding specific products in such directories is facilitated through grouping of projects, according to categories, and searching and sorting services. Unfortunately directories like Freshmeat and Sourceforge fail to exhibit generality in how they collect and present information. That is to say, they don't cooperate on content nor format, so as a result the same project can be described completely differently depending on which directory one visits.

While software directories facilitate discovering and attaining software, they offer little help when it comes to deploying it. The exact procedure is still up to the individual project. That being said they facilitate the update process somewhat. As a project in a software directory releases updates, they can at least be sought out by visiting the project's directory entry and examining its release list. Directories may even provide an RSS feed so users can be notified of new project releases, at least Sourceforge has this option [32]. In either case, it is left to the user to acquire updates and deploy them.

2.2. The PyPkg vision

conduit

> *a passage (a pipe or tunnel) through which water or electric wires can pass; "the computers were connected through a system of conduits"*
>
> *- Dictionary.com [10]*

The preceding discussion should provide some insights on the situation that we mean to improve. To recapitulate, we are interested in facilitating distribution and deployment of software in general. In this respect we see the former as a supportive function to the latter, since software must first be distributed to users before they are able to deploy it (barring direct access to the "master" copy, obviously). That being said, the distribution process is of course relevant outside the deployment context. What we are dissatisfied with when it comes to current conventions, regarding both distribution and deployment, is essentially the lack of a *uniform* method. Software is made available in so many ways (both with regard to reachability and format), and as a user one is exposed to great variability in terms of deployment approach (depending on one's platform, granted). This

is something we believe is possible to rectify with a common system, hence PyPkg.

2.2.1. The conduit concept

Having applied considerable thought to the distribution/deployment problem throughout this project, we've come to the conclusion that it is best solved by a **conduit**-like system (Figure 2.1). To elaborate on this idea, one could say the Internet already presents a general conduit for transferring digital content between parties, we propose a specialized conduit for software. The utility of this special conduit is that it is capable of integrating deployment with transfer of software, meaning that it extends the "Internet conduit" beyond simple distribution. By integrating distribution with the deployment process, we effectively attain a uniform distribution method as a byproduct, being a requirement for seamless integration. Like the lower-level conduit enabled by Internet protocols our software conduit is intended to be completely general with regard to platform and content (provided it falls in the software category of course).

On the basic level software is a type of (digital) content; when treated in a totally generic manner, as a sequence of bytes, the Internet serves as an unbroken conduit for moving it from A to B. Thanks to the generic nature of the Internet Protocol Suite [8] content can be effortlessly distributed between parties, regardless of their respective platforms or how many intermediate network links it must pass through. We can see no intrinsic reason why this shouldn't be true of the general software transfer process as well[1], conceptually a platform could exist where downloaded software could invariably be applied directly without any need for installation. Since the general platform is a lot more complex than this, software content must receive special consideration.

Thus we envision PyPkg as a special conduit that treats software as "rich" content, that must be deployed in order to truly complete the transfer process. Since the system is to model the unhindered flow of a conduit, it must be capable of conducting deployment in a completely seamless manner, regardless of involved complexity (e.g., requirements related to installing software). This can be likened to how Internet protocols let two parties open a seemingly direct connection without worrying about the physical path, a number of interconnected nodes, connecting them.

To make this notion more concrete, PyPkg is to provide facilities for software distribution and deployment that are capable of collaborating in order to form a conduit for transferring software. These facilities should be generic with regard to type of software and platform.

Since we mean to establish a common method our distribution facilities must be applicable to general distribution purposes, corresponding to the role traditionally filled by project Web pages and software directories, in addition to supporting deployment through PyPkg. Hopefully this will compel developers to agree on PyPkg as a common distribution system, while also paving the way for our software conduit concept. Furthermore, PyPkg should present a certain distribution network for all software reachable through the conduit, that can be used both to access and discover software. This should serve as a kind of generic software directory, conceivably it can act as a source of information for special-purpose software directories, thereby avoiding the problem where directories operate with separate representations of indexed software.

The deployment facilities are to present a streamlined method for deploying software

[1] By software transfer we are thinking of how a piece of software is obtained from a provider and deployed on a target system, ready for use.

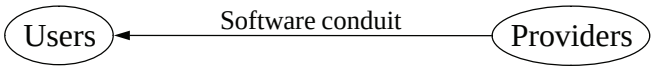

Figure 2.1. The software conduit

on target systems. This means that they must handle all requirements when software is being installed; notably, any inter-software dependencies should be handled for the user. In addition PyPkg's deployment functionality is to include support for updating and uninstalling installed software. Given the direct connection to the distribution layer presented by the conduit, it is only natural to automate the update process, especially considering the security aspect discussed in Section 2.1.2.2.

2.2.2. Precursors to the conduit concept

Although the conduit concept may be novel, the idea is not entirely original. Inspiration was to a great extent drawn from the APT style of Linux package management, via the derivative Portage system. Such systems establish a type of conduit through which users can obtain (deploy) software, in what is to the user a single operation. What characterizes the APT model in this respect is that software is collected, in packaged form, in central repositories. Compatible client programs (package managers) can contact such repositories and thereby gain access to all data they need to process offered software on behalf of users. As such, a conduit is effectively established between package repositories in one end and clients (doing the user's bidding) in the other.

Package management systems like APT and Portage are able to offer their users some great functionality; notably, when installing a package, other packages depended on are also installed (cf. Section 2.1.2.1). In addition, installed packages can be updated either on an individual or a global basis (by considering all installed programs) as new versions are released. Traditional Linux package management systems take on a more fundamental responsibility as well; there being no standard system level deployment service on Linux (like Windows Installer [28]), this has come to be the responsibility of package management systems. Installations are recorded in databases (these differ between systems) so they may be managed by the user (e.g. uninstalled).

Although the APT model serves as a good example of how software deployment may be streamlined in a heterogeneous environment (software is collected from many different providers), it is a greatly flawed solution. It simplifies the problem by being highly specific, i.e. each package management system establishes its own independent distribution channel. That is to say, APT has its own package repositories, Portage has its own package repositories etc. To the user the distribution channel appears streamlined, but in most cases it will require an extra adaptive step.

The centralized (and specialized) repository model does not reflect how software is normally developed and distributed in a completely decentralized manner. Providers cannot simply publish their software in one location, say in the "Downloads" section of the project website, for it to enter the distribution channels of APT-style package management systems. Each release must be packaged in the corresponding format (.deb for APT/dpkg [49]) and placed in a centralized repository, a procedure that must be repeated for all different package management systems one

intends to support. The result is that providers tend to stick to the tried and true solution of releasing their software in a generic manner, as discussed in Section 2.1.1, leaving it to volunteers to package software for various package management systems (the adaptive step).

The Windows Update [52] service is another point of reference for our conduit concept, albeit even more specialized than APT-style package management. This service is specifically directed at automatically deploying updates to Microsoft Windows system components on users' systems, as they are published centrally. Windows Update establishes an unbroken conduit between the provider (Microsoft) and end users, and as such it is very good for what it does (transferring/applying updates).

Chapter 3. Model

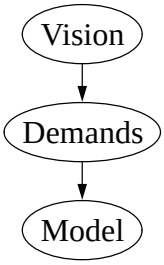

Figure 3.1. From vision to model

The concept of a conduit for facilitating software transfers is something we believe strongly in, the previous chapter helped define how we intend this notion to manifest itself. Still, the product vision serves as a highly abstract definition and doesn't quite reflect the complexity of creating a corresponding software system (which is not the vision's purpose either). There are many obstacles that such a system must cope with; there's a lack of formal conventions for distributing software on the Internet, software tends to impose various requirements with regard to installation, platforms vary greatly in their support for software deployment etc. The PyPkg model defines in exhaustive detail how the system is to solve these problems.

Considering the involved complexity, it is imperative that we are systematic in designing PyPkg. To this end, we take a certain approach. First of all the high-level product vision is translated into a set of discrete demands on PyPkg, this serves as our concept of the distribution/deployment problem so to speak. Moreover, this concept represents a simplification of the full problem, at least in early stages, in order to simplify development. That is to say, for each development iteration we identify a set of discrete demands from the product vision, that is deliberately limited with regard to the full vision. The next iteration identifies, and takes on, new demands etc. From such a set of demands it's possible to deduce a model, that details (to differing degrees) how the corresponding solution is to manifest itself. Figure 3.1 illustrates this process of refining the vision into a model.

Before discussing PyPkg's design in full detail we will provide some background on how we derived the high-level architecture, in the context of involved problems. As per our design method, we've decided on a somewhat simplified problem with regard to the vision, where the system only concerns itself with two deployment tasks: installation and uninstallation. Otherwise the model is quite in line with the envisioned distribution/deployment facilities (described in Section 2.2.1).

The in-depth discussion on the design is divided into three sections. Our model consists really of two main parts (sub-models) corresponding to the two ends of the software conduit, distribution and deployment, but the scheme for handling software dependencies (refer to Section 2.1.2.1) is shared between these two and so discussed separately. The division into two

main parts was natural since PyPkg's distribution network is to exist independently of the deployment facilities. There is a need for collaboration on the distribution network's part to enable the PyPkg deployment process, but it is done in a generic manner that can even be exploited by other systems. In conjunction with the discussion of each of the two sub-models, the corresponding set of demands is presented.

3.1. Considerations for an automated software transfer process

As described in Section 2.2.1, PyPkg is meant to provide a generalized software conduit on top of Internet transfer mechanisms. The Internet solves the problem of transferring data between heterogeneous parties for us, but there's still the problem of heterogeneity in how software is presented for distribution and how it is deployed on target systems. This implies that in order to automate the transfer of software on a general basis, a higher level architecture must be imposed on the process, covering all relevant aspects.

3.1.1. Involved problems

In order to at all enable a homogeneous transfer process, a common method for publishing distributed software must be decided on (given the lack of a standard approach). Navigating an arbitrary project Web page in order to find offered software may prove difficult even for a human, if the project in question has indeed released any at all. When distributing through an automated system software releases should be subject to formal rules, thereby eliminating uncertainty related to this process.

The actual deployment of software on individual systems can get quite involved, depending on the particular software and target platform. Some problems related to deployment are general in scope, while others emerge through dealing with particular platforms. A rule of thumb for us when considering automated deployment is that all steps that would have to be performed manually are to be encompassed by the deployment process. This includes handling any requirements imposed by software with regard to installation.

A particularly problematic type of requirement in conjunction with installing software is dependencies on other software (as discussed in Section 2.1.2.1). It is possible to ignore this problem altogether design-wise and instead require that providers bundle their software together with any dependencies, but such an approach would cost us some flexibility. Apart from extra effort on the providers' part and added space requirements (due to bundling), it is problematic in conjunction with software updates. Any updates to dependencies would have to be bundled with new versions of dependents, in order to replace older versions installed on users' systems. Such an update model would be especially inhibitive with regard to Free Software projects, that are prone to reuse components developed independently by other projects. Free Software is of particular relevance to us given our focus on the Linux platform, with its affinity to this development philosophy.

In order to retain flexibility we've decided on an explicit dependency scheme. Software distributed through PyPkg should be allowed to depend on other software (distributed through PyPkg), as long as such dependencies are explicitly stated. Consequently, our distribution method should provide the means for representing dependencies between different software. Also, with automation in mind, PyPkg must concern itself with satisfying such dependencies

when installing software: when a software product is transferred across the conduit to a user other software may be included as dependencies.

It should be noted that despite motivating the flexible (modular) dependency scheme, we're leaving out support for automated updates at this stage. This will hopefully be addressed in future development.

For our conduit to support heterogeneous target systems it must present a layer on top of system-specific deployment mechanisms, that serves as an interface to all supported deployment functionality. A homogeneous deployment layer should be appreciated especially by users who deal with multiple operating systems, given that they can (conceivably) spend more time applying software and less time installing/maintaining it. That being said, when prototyping PyPkg it makes little sense to prioritize multi-platform support.

3.1.2. Achieving a generalized conduit

When designing our system we look to other conduit-like systems such as APT and Windows Update (see Section 2.2.2 for a discussion of sources of inspiration), with the difference that we are not developing a solution for a particular platform nor a specialized purpose. Our design most resembles the APT-style package management model, but for us it is imperative to meet the *general* usage case. That is to say, we'd like for our system to be the common method for developers to offer their software to users, corresponding to the role of Windows Installer packages (.msi) when targeting Windows users for example.

Designing such a system is, as might be expected, easier said than done. We already know that the PyPkg model is to consist of two top-level aspects, *distribution* and *deployment*. The distribution aspect is to present facilities for publishing software so that it can be reached through the conduit, i.e. it must allow providers to present their products according to a specific protocol. The deployment aspect is concerned with letting users deploy software published through PyPkg, and uses the distribution protocol to query the products of individual providers. Hence, the distribution aspect plays a supportive role relative to the deployment aspect. Even so, the protocol needn't be particular to PyPkg. In fact, in our view all parties would be best served by a common protocol for publishing software. Regrettably, there exists no commonly agreed-upon alternative, instead we will have to settle on a distribution model in the hope that it will find general acceptance.

As a whole, then, the PyPkg model must represent an environment for establishing a conduit between those offering products (**providers**) and those interested in said products (**users**). We conceive this environment as populated with representations of the two types, where one kind assumes a passive role (providers) whereas the other (users) is active. This role division should enable a push/pull model, where providers can publish their products once (push) and interested parties query this information directly (pull). In practice this environment should manifest itself as a "database" of distributed software products, represented in terms of the PyPkg **protocol**, which is accessed by PyPkg **agent** programs, acting on behalf of users.

In order to involve providers directly in the distribution process as much as possible, the model must allow for decentralization in the distribution aspect. The "database" of distributed software should actually be shattered across individual Web resources, typically corresponding to project websites. This is because we want distribution through PyPkg to primarily integrate with ordinary distribution mechanisms (i.e., avoid creating another distribution channel). It

would be unrealistic to expect instant widespread acceptance of our distribution model, but this is the base behaviour we should strive to accommodate.

3.2. Distribution

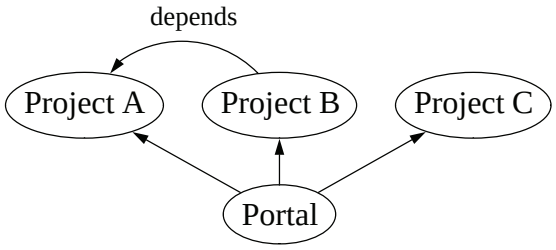

Figure 3.2. The PyPkg distribution model

The distribution aspect of our model pertains to the envisioned PyPkg distribution network (Section 2.2.1). Based on the ruminations in Section 3.1.2 we've given some thought to how to establish a network of decentralized representations of software resources, in a generic manner. What seems the sensible option to us is a Semantic Web [29] approach, which is how the World Wide Web inventor Tim Berners-Lee envisions that machine-processable information be shared across the Web [6]. The Semantic Web initiative seeks to augment traditional Web technology (HTML), which provides facilities for describing format rather than content, with means to convey meaningful information (through semantic assertions) between machines (e.g. computers). This approach of meshing with current Web architecture is particularly appealing to us, in that it should ease the transition from traditional Web-based software publishing.

In practice the Semantic Web provides us with a common, general, language for describing distributed resources. This language gives us the means to devise the PyPkg protocol (Section 3.1.2), for publishing software in a manner that enables a general software conduit. More specifically, we've decided to compose the PyPkg distribution layer from Semantic Web representations of individual software providers and associated resources, described according to the PyPkg protocol.

The demands

Having outlined the basic model for the PyPkg distribution layer, we present the concrete set of demands. These serve to flesh out the informal demands presented in the discussion of the product vision, in Section 2.2.1.

1 The PyPkg distribution layer must present a network that allows for navigation of software resources

2 The PyPkg network is to consist of decentralized representations of software resources

3 PyPkg must expose a Semantic Web protocol that governs the description of software resources

4 The PyPkg distribution protocol must support general publishing of software releases, analogous to how it's done via Web pages

5 The PyPkg distribution protocol must give access to software packages deployable by PyPkg agents for software releases

6 The PyPkg distribution protocol must make it possible to express inter-software dependencies

The last demand has to do with the dependency scheme which affects both the distribution and deployment aspects of the PyPkg model, described in Section 3.4.

The model

Figure 3.2 shows a condensed version of the PyPkg distribution model. As can be gleaned from the diagram, the model employs a hybrid approach where decentralized software project representations are accessed through a central portal. The portal meets the first demand by presenting a public index of the distribution network, organized in terms of *providers*. Each such provider is associated with a set of *projects*, each project representing a specific software resource (product). The second demand is accommodated through an indirection scheme, where provider descriptions in the portal index simply refer to project representations that can reside anywhere on the Web (i.e., they are decentralized). This means that providers are (as far as the model is concerned) responsible with hosting and maintaining representations of distributed software themselves, reflecting how distribution traditionally revolves around autonomous project websites.

The providers (in the portal index) and associated projects are described through a special Semantic Web protocol (as per demand number 3). Now, the Semantic Web itself is an umbrella for several languages intended to facilitate sharing of data on the Web between machines (computers). Of particular interest to us, *RDF* (Resource Description Framework) has emerged as a W3C recommendation [45]. RDF provides the basic building blocks for sharing data, through the description of distributed resources and relations between them. As such, the PyPkg distribution protocol is expressed as an RDF vocabulary (containing special terms relevant to the software distribution problem).

Software is published according to this protocol by adding descriptions of software releases to RDF project representations, such descriptions give access to data both relevant to general distribution purposes (demand 4) and to PyPkg agents (demand 5). Essentially each release description states general metadata about a release, such as its version, and also refers to one or more downloadable packages. There should at least be a package in the PyPkg format (which PyPkg agents are able to utilize), but additional packages can be offered in arbitrary formats (e.g., zip or tar archives) for general distribution purposes. Due to the general nature of RDF and the PyPkg protocol, project releases published in this way can really be harnessed for any intent, not just PyPkg purposes. For instance, the RDF data can be used to automatically generate release announcements on project Web pages.

The diagram also shows a dependency relation from project B to project A, this is to illustrate that the protocol has support for modeling dependencies between different software (corresponding to demand 6). This happens on the project release level, such that a release description can state dependencies on software offered by other projects, which need to be satisfied upon installing the release. Such dependency declarations are particularly directed at PyPkg agents, but the dependency scheme is designed in such a way that it can serve as a general method for formalizing software dependencies.

A proper in-depth treatment of the distribution model is given in subsections 3.2.2 and 3.2.3. The former details the base PyPkg distribution model, which is centered around decentralized project descriptions, while the latter describes the portal which provides a central access point to the network of projects. Before we get to that however, an explanation of the RDF fundament on which our distribution model rests.

3.2.1. Some words on the use of RDF

Basically, what RDF provides us with is the means to describe distributed resources, represented by URIs (Unique Resource Identifiers), through binary relations to primitives (RDF literals) and/or other resources. Each such relation serves to assert a property of the described resource. RDF itself isn't a concrete language but an abstract datamodel, for describing data on a *triple* form. According to this model, knowledge (of resources) is aggregated through successive evaluation of such triples.

3.2.1.1. The RDF datamodel

An RDF triple consists of the components *subject*, *predicate*, *object* [25]. The subject is the URI of the resource being described, the predicate, also a URI, denotes a certain property of the subject, while the object is the value (either a resource, identified by a URI, or a primitive) of this property. Effectively, an RDF triple states a relationship, indicated by the property, between the subject and the object. A simple example of a possible RDF statement (triple), **(thesis, http://www.simula.no/rdf/author, "Arve Knudsen")**, states the author of this thesis.

The triple is the basic mechanism for conveying meaning in RDF, but the amount of information that can be expressed on this form is limited (to binary relationships). Rather, a collection of RDF triples is normally interpreted as a graph. Nodes correspond to subjects and objects, which are connected by directed arcs corresponding to predicates. A graph of this kind will then represent a comprehensive bank of encountered knowledge about contained resources. When compiling a graph, individual statements can come from entirely different sources. Knowledge from statements is simply added to the graph as they (statements) are evaluated, a single resource (identified by a URI) can even be described by different sources (statements referring to a resource are merged without regard to their location). Figure 3.3 displays the graph representation of the example triple.

3.2.1.2. RDF Schema

In order to attach some meaning to information expressed by RDF graphs, there must be a certain *vocabulary* involved. That is to say, when stating a property of a resource, the property must have a specified semantics. Otherwise the statement won't tell us much, i.e. the meaning of the property in question is unknown to us. As properties in RDF are identified by URIs they can be recognized by the processing application, and translated into corresponding semantics (if they are indeed recognized). For instance it should be fairly obvious to a human reader what the `http://www.simula.no/rdf/author` property of the previous example signifies, while a typical RDF-processing program would match it directly against known properties.

When it comes to sharing information between parties a well-specified vocabulary is especially pertinent; a vocabulary effectively presents a protocol, which consumers come to

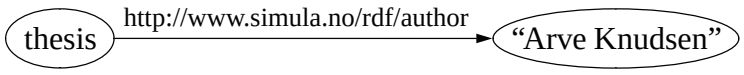

Figure 3.3. Graph representation of example RDF triple

rely on in order to make sense of data presented to them. In a typical RDF scenario vocabularies from different sources are also mixed and matched, so to reuse well-known semantics. This is accommodated by the use of URIs for RDF properties, so name clashes and resulting ambiguities are avoided.

Even though RDF descriptions can be parsed without any formally defined vocabulary, there exists a special layer for this purpose: *RDF Schema* (RDFS) [46]. RDFS vocabularies can be used to validate RDF sources, but in general they are used to document semantics. If an RDF processing application validates against RDFS definitions it can catch the use of unknown properties or unintended use of properties.

An RDF vocabulary defined in RDFS can establish a class hierarchy, as known from the object-oriented programming paradigm [51]. RDFS lets you define resource classes, themselves resources, and associated properties. RDF properties are really free-standing, but when defining a property it can be associated with a class by specifying its *domain*. The domain is actually itself a property of instances of class `Property`, which is used to define new properties, and refers to a class resource. Thus, new properties can be added for a class at any time, anywhere. Classes can also derive from existing classes, in order to be compatible with base class properties.

The primary reason for defining an RDF class (in RDFS) is to constrain the set of properties that can be used to describe a type of resource. By doing so for a set of classes, one may precisely model a domain of resource types (i.e., an ontology). For our purposes this is relevant for modeling the roles that partake in our distribution layer, creating a common RDF vocabulary so software providers may describe themselves to interested parties (PyPkg agents for instance).

For a relatively simple example of RDFS definitions one may refer to the PyPkg RDF vocabulary, in Appendix A.

3.2.1.3. Concrete syntax

The abstract RDF graph model needs a concrete syntax in order to be machine-processable, i.e., graphs must be serialized according to some language. There exist several serialization syntaxes for RDF, of which RDF/XML [47] is the most common. RDF concepts map cleanly (albeit not uniquely) to XML terms: element names, attribute names, element contents and attribute values. RDF vocabularies (as defined using RDFS) translate to XML namespaces.

RDF/XML is our syntax of choice for information exchange in the PyPkg distribution layer. There exist more compact/“intentional” syntaxes for expressing RDF statements, for instance Turtle (Terse RDF Triple Language) [5] makes for a very readable and at the same time expressive syntax. However, XML is by now a commonly understood declarative language in the IT world and should be a safe choice for both computer-aided and manual reading and editing. In fact, the PyPkg distribution layer is in practice composed of RDF/XML documents pertaining to software providers (the portal index) and their projects (hosted separately).

Figure 3.4 expresses the same information as Figure 3.3, transformed into RDF/XML.

```
<rdf:Description rdf:nodeID="thesis">
   <simula:author>Arve Knudsen</simula:author>
</rdf:Description>
```

Figure 3.4. Example RDF triple in XML syntax

3.2.2. The PyPkg RDF model

The PyPkg distribution layer, organized in terms of software providers, is modeled with an appropriate object-oriented RDF vocabulary. As such, we will be required to decompose the problem domain into classes of resources. A sufficiently descriptive and well designed model can be complex enough to develop in itself, in addition it is a stated goal that the model should strive to integrate with ordinary distribution mechanisms. The latter is important when it comes to achieving acceptance among those responsible for developing and distributing software, we'd like to ease their burden rather than adding to it. So in order to avoid reinventing the wheel, and possibly also leverage existing momentum, it is a good idea to evaluate any previous efforts in the field.

3.2.2.1. DOAP

We've discovered at least one RDF vocabulary with goals compatible with ours, that aims to provide a common framework for self-describing software providers on the Web. The *DOAP* (Description Of A Project) vocabulary [15] is centered around the project entity, as indicated by its name. A DOAP project corresponds to our concept of a software provider, with the distinction that a DOAP project directly represents a single product. Apart from this, the vocabulary is meant to provide descriptive terms of such a general nature that interested parties can pull metadata directly from software projects, rather than it being scattered across different sources (e.g., software directories).

The main entity in the DOAP vocabulary is the `doap:Project` class (`doap` denotes the vocabulary). A number of properties are defined for this class, that are deemed generally interesting when describing a software project. Prominent properties include `doap:name`, `doap:description` and `doap:homepage`. Figure 3.5 shows a somewhat simplified representation of the DOAP class hierarchy. For full disclosure of the vocabulary see [16]. There is some support for the distribution aspect in DOAP, but it is currently rather weak and restricted to describing the current project release. Such a description (through the `doap:release` property) can be used to provide a link to a downloadable package but not much more.

A project supports the DOAP model by hosting a corresponding DOAP document. The model is fully decentralized, and there is no particular way of registering a DOAP-enabled project so that it can be discovered by others. The DOAP author recognizes that there may be a need to uniquely identify a project however, for referral by others. The raw URI of the project description is not a good idea for this purpose since it is subject to change, instead a project is in the DOAP context uniquely identified by its current and previous homepages [14]. For this to work a project may never acquire a homepage previously owned by another. The `doap:homepage` and `doap:old-homepage` properties are used to describe the current and

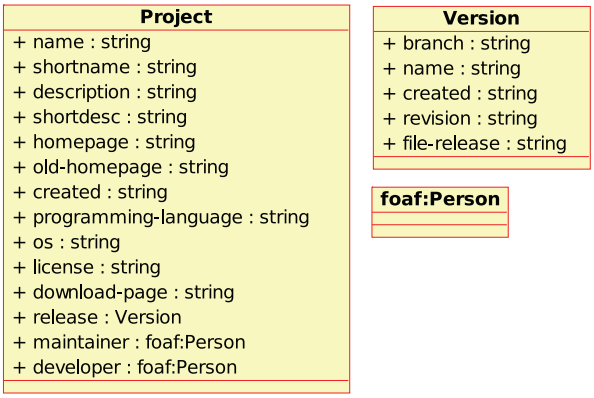

Figure 3.5. DOAP class hierarchy (simplified)

any previous homepages, respectively. Every previous homepage of a project should be listed using `doap:old-homepage`, so that a processing agent can recognize a project even based on a reference to an outdated homepage.

3.2.2.2. PyPkg: extending the DOAP vocabulary

As far as we can tell DOAP is a very good design for what it does. It represents a sensible approach for publishing projects online, and as such it has recently been adopted by the O'Reilly CodeZoo initiative [31], a software directory strictly dedicated to components and development utilities. It lacks real support for deployment agents, though. That being said, a hallmark of RDF is the ease with which one can combine/extend vocabularies. DOAP itself imports the Friend Of A Friend (FOAF) [13] and Dublin Core (DC) [4] vocabularies, in order to employ established "standards" for certain types of metadata (for instance DC defines a `dc:creator` property).

For the PyPkg distribution layer DOAP forms an excellent foundation through which projects may express generally useful metadata, the additional needs of deployment agents can be addressed through some PyPkg-specific vocabulary extensions. Those familiar with the resulting vocabulary, DOAP/PyPkg, will be able to both extract general project metadata (courtesy of DOAP) and determine how to deploy a project's software releases: how may a deployable package be acquired, what other software is required (dependencies).

We will now look at what is concretely required to extend DOAP with deployment support. To recapitulate, the underlying intention is to create distribution facilities capable of providing autonomous agents with all information necessary to deploy distributed software. Refer to Section 3.3.1 for a definition of the PyPkg agent concept. A useful rule when dealing with such agents is that they possess limited reasoning powers, and gain necessary knowledge mainly by applying simple logic to input data (RDF in this case). To this end, it is the PyPkg distribution layer's responsibility to provide sufficient information so agents may carry out their tasks successfully.

Whereas humans can utilize tacit (implicit) knowledge or synthesize new knowledge through experience, agents of the PyPkg type need unambiguous facts (assertions) that they can act upon. As an example, when installing a software package the user may need to take several requirements into consideration. For one, the software may be bound to a certain platform (Windows 2000 or newer, perhaps). Two, it may depend on other third party software during execution. Both requirements may be communicated informally by the provider, if at all. For instance,

it is possible that the user is first able to deduce requirements from error messages when trying to run the program. If an agent is to perform this task for the user it must receive such information upfront. In our case, RDF is the transport for knowledge (or "actionable information") intended for agents.

As we know, the project is the central DOAP entity. The `doap:Project` class has many different properties that are of little relevance for deployment purposes, reflecting how DOAP is mainly aimed at consolidating project metadata. The distribution aspect is served by the `doap:release` property, which can be used to publish the current software release of a `doap:Project`.

A concrete release is described by a `doap:Version` instance. This is a relatively barebones representation that can communicate some typical properties of a release: when was it released (`doap:created`), what's its revision number (`doap:revision`), does it belong to any release branch (`doap:branch`), is it assigned a particular name (`doap:name`), how can it be acquired (`doap:file-release`). Such metadata are generally interesting to consumers, but far from sufficient for deploying the release in an automated manner. The newly added `doap:file-release` property should at least give access to a deployable package, but only as a raw URL (meaning that the deployment method must be inferred by other means, by inspecting the file type or its contents if its an archive).

The `sw:Package` class

The `doap:file-release` property proves a good starting point for adding more extensive support for the deployment aspect to the DOAP model. As it stands, `doap:file-release` is rather underspecified. It can refer to any type of resource, although it's most likely intended for URL strings. A more useful approach would be to describe the downloadable deployment resource explicitly, for processing by agents. To this end, we've decided to add a class `sw:Package` for describing deployable software packages. Rather than "overloading" the `doap:file-release` property to refer to package descriptions, we've added a property (of `doap:Version`) specifically for this purpose: `sw:package`. The latter is restricted to referring to `sw:Package` instances only.

`sw:Package` is intended as an entirely generic base class that can be subclassed to indicate particular kinds of packages, for instance the PyPkg package format. Properties are defined for this class that are applicable to all types of packages: `sw:size`, `sw:download-url` and `sw:md5`. The size is generally good to know when downloading a file (package), and even crucial if the agent supports resuming incomplete transfers. `sw:download-url` serves a similar purpose to `doap:file-release`, although the type of the file should be known in this case. `sw:md5` is used to present the MD5 [50] checksum of the file, for verification of the downloaded file. PyPkg uses MD5 as both a security measure (to detect whether packages have been tampered with) and to avoid file corruption (in particular, when resuming transfers). A special `sw:Package` subclass is used to describe PyPkg packages, `pypkg:Package`.

Declaring dependencies

As it is a stated goal that PyPkg deployment agents be able to handle inter-software dependencies, the distribution model must do its part by letting dependencies be declared for project releases. This follows from the requirement that the distribution layer disclose all information necessary for an agent to carry out its task successfully. Dependencies on third party components is

a common problem when deploying software on Linux in particular, and an installation can't be considered successful until all such dependencies have been met. Currently we have a relatively simple model for declaring dependencies, it can only happen on the release level. A more realistic approach would perhaps reflect that the set of dependencies may vary between package types, a Linux package is likely to have more dependencies than a Windows package for instance, but at the time being only Linux PyPkg packages are relevant to our work.

In order to support dependency declarations in project descriptions, we've added a property `sw:dependency` for the `doap:Version` class. The property may be listed several times for one release (once per dependency), and refers to instances of class `sw:Implementation` (another extension). The reason for introducing the `sw:Implementation` class is to represent implementations of software "interfaces" (refer to Section 3.4 for an explanation of the interface concept). An `sw:Implementation` instance, when used in the `sw:dependency` context, must declare the interface it implements and the corresponding interface revision, through the `sw:interface` and `doap:revision` properties respectively. The `sw:interface` property must refer to a PyPkg project (as a URI), each project is also considered an interface. By that we mean that as each project is linked to *one* product, a project is considered to present an interface through this product (which other software may depend on).

When a deployment agent encounters a dependency declaration, it will know the URI of the interface and which revision of said interface is being depended on. In order to do something about the dependency however, the abstract dependency specification must be resolved into a concrete implementation (a project release). In order to do so the agent must be able to locate software implementing a compatible revision of the interface (being depended on). For this to work the project corresponding to the interface is required to implement all versions of its interface, and should be reachable through the `sw:interface` property of the dependency specification (an `sw:Implementation` instance). This is an important point. For the dependency resolution process to work, agents must be guaranteed stable URIs. If the project for an interface depended on can't be found due to a dangling URI, the agent will likely fail in resolving it into a concrete implementation.

In order to be able to guarantee stable URIs, we state that all inter-project references in PyPkg project descriptions must go through the central portal. The completely decentralized DOAP project identification scheme is simply too weak for our purposes. It is not enough for a PyPkg agent to know how to recognize a project, it must also be able to reach it. To this end, the PyPkg portal provides constant URIs for all PyPkg project descriptions (Section 3.2.3). If a project description needs to reference another project it must do so through a URI prefixed with the portal's base URI.

Connecting interface versions to implementations

By definition, all PyPkg projects expose an interface which other projects may depend on. This interface is implemented by project releases and may evolve over time, that is, each release corresponds to a certain version of the interface. Since a project's interface effectively represents a public contract, it is important that its relation to actual releases by the project is formalized.

To this end, the PyPkg model requires that each project release (`doap:Version`) declares which version of the project's interface it implements. For this purpose we've extended the `doap:Version` class with a property `sw:implements`, which refers to instances of class `sw:Implementation`. `sw:Implementation` instances in this context are taken to refer to

the project itself (`doap:Project` is declared as a subclass of `sw:Interface`), but must state which revision of the interface is implemented (using the `doap:revision` property).

The gist of this scheme is that when an agent is looking to resolve a dependency on a certain version of an interface, it need only look up the corresponding project and see which releases implement it (the interface version). This is highly practical in that the owner of the interface provides direct advice on how dependencies on it may be resolved. A dependency on an interface version that has never been implemented is considered an error.

The resulting vocabulary

The vocabulary resulting from our extensions to DOAP, `sw` for "software", is displayed as a class diagram in Figure 3.6. Although RDF properties are not explicitly defined as part of a class, we show each property in the vocabulary as a method of the class it is associated with through the domain attribute (see Section 3.2.1.2 for an explanation).

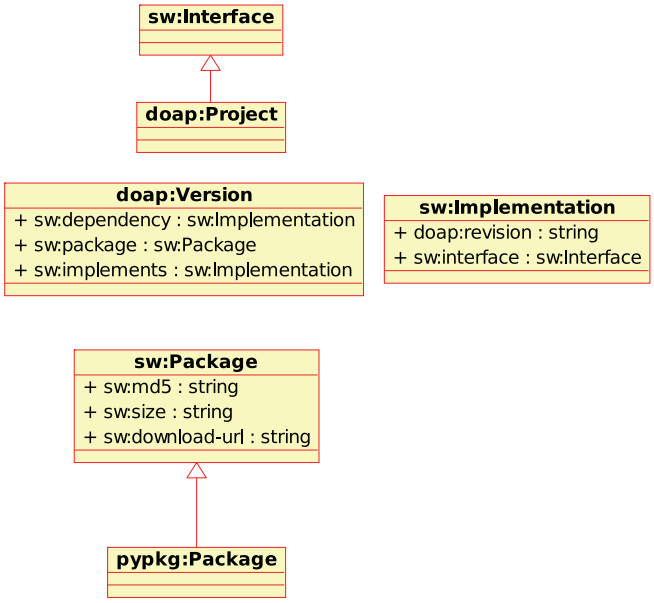

Figure 3.6. PyPkg RDF model (DOAP extension)

3.2.3. The PyPkg project portal

Cool URIs don't change

- Tim Berners-Lee [7]

The central PyPkg portal presents an index, in RDF, of all (registered) project descriptions, and also a stable URI namespace for projects. The index only references individual project descriptions (through URIs), so even though the portal is centralized project descriptions are not. Rather than a flat list of projects, the index is organized in terms of *providers*. The notion of providers in this context is introduced in order to group projects that are controlled by the same

organization. It is far from unusual that the same organization is responsible for a number of software products, each of which corresponds to a PyPkg project.

In order to present such provider entities through the portal's RDF index, a special vocabulary has been devised, `swidx` for "software index". It is quite simple: a class `swidx:Provider` exists to describe PyPkg providers, and can be used with properties `doap:name`, `swidx:id` and `swidx:project`. The `doap:name` property is only used to disclose the full name of the provider, which can be useful for presentational purposes. The provider's `swidx:id` must be a unique identifier within the portal's provider namespace; provider identifiers are in turn used to generate unique identifiers for projects, which means that project names only need to be unique in relation to other projects from the same provider. The `swidx:project` property only lists a project identifier, typically corresponding to a short form of the project name. The `swidx` vocabulary is shown as a class diagram in Figure 3.7.

The full URI of a project can be found by combining the portal's base URI with the provider identifier and the project identifier (separated by slashes). As such, one needn't look up a project in the portal's index in order to determine its URI. The portal uses a URI namespace scheme where each project receives a URI that is a combination of the portal's URI, its provider's identifier and its own identifier: **<portal-URI>/<provider-identifier>/<project-identifier>**. Exactly how the portal organizes access through such URIs to project descriptions is not important with regard to the PyPkg model; presumably the portal may either provide an indirection service or perhaps cache project descriptions as they are updated (in order to secure timely access).

For a project to be part of the PyPkg network it must be registered with the portal. This takes place by adding the project among the corresponding provider's projects, a provider description must be added if not already part of the index, after choosing a suitable identifier for the project.

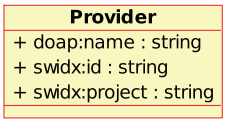

Figure 3.7. PyPkg portal RDF model

3.3. Deployment

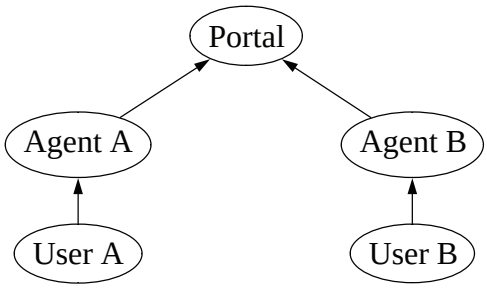

Figure 3.8. The PyPkg deployment model

The deployment aspect of the PyPkg model corresponds to the user end of the envisioned software conduit (Section 2.2.1), which means that it represents the user's connection to the PyPkg distribution network. The implication of this is that it is responsible for providing an automated method for deploying software distributed through PyPkg. This method manifests itself as agent programs that act on behalf of the user, as illustrated by Figure 3.8: PyPkg agents collaborate with the distribution layer, via the central portal, in order to automate deployment processes.

Designing the deployment aspect of PyPkg presents quite different challenges from the distribution aspect, since it is concerned with behaviour (of agents) as opposed to static data. The previous section was quite sufficient for documenting the distribution model, but the deployment model is of such complexity that it is best detailed in a dedicated design document. We've in fact maintained a design document as an aid in implementing the deployment subsystem, so we might as well draw on that in this text rather than repeating ourselves (and violate an important software engineering principle).

As such, we take a different approach from the section on the distribution model; we aim in this text to present PyPkg deployment functionality in a high-level manner, using the design document as the formal reference. As opposed to the design document's strict architectural orientation, a process-oriented view is employed. Specifically, we see the deployment model as pertaining to two top-level processes: installation and uninstallation. Ideally there would be a third process for updating installations, as per the vision (cf. Section 2.2.1), it is left for a future version of PyPkg however.

The PyPkg deployment processes are discussed in subsection 3.3.2, in relation to demands on the deployment subsystem. Before fully delving into the design, we will provide some background on the agent concept as used in the PyPkg context, considering its central role in the deployment layer.

3.3.1. The agent concept

Our agent concept is based on the one defined in [40, p. 32]. Basically, we see an agent as a rational (acting optimally with regard to its knowledge) program which acts on behalf of a human entity. Agents are designed for the purpose of accomplishing a certain goal. In our case it is that of deploying software packages distributed through the PyPkg network on user systems. In the Artificial Intelligence context, our type of agent is relatively simple when it comes to problem-solving, most resembling a "knowledge-based agent" [40, p. 195].

The defining trait of a knowledge-based agent is that it bases its actions on a knowledge base, i.e., accumulated knowledge aids it in making rational decisions (that are most likely to lead to a successful result). In the PyPkg system, the distribution network serves as a global knowledge base. PyPkg agents are able to exploit knowledge contained therein through simple logic; RDF can be interpreted as a type of first-order logic [9], and as such agents can infer truths by combining assertions from the knowledge base.

A non-trivial type of problem that is solved through logical inference in PyPkg agents is determining dependency chains, i.e. what software to include in order to satisfy dependencies when installing a software package. Specifically, when considering a project release for installation one will look at the release's asserted dependencies on versions of software interfaces. For each unsatisfied dependency (dependencies satisfied by the host system are ignored) the agent should be able to find, in the knowledge base, at least one assertion that it is being implemented by a

project release[1]. Logical entailment tells the agent that it must include the preferred (currently equivalent to newest) implementing release in the dependency chain for the installation process, basically corresponding to the rule: *depends(X,Y),notSatisfied(Y),implements(Z,Y),preferred(Z) -> include(X,Z),* where X is the project release being considered for installation, Y is a dependency of X and Z is a project release implementing Y[2].

3.3.2. The design

When designing the PyPkg deployment layer, we factored common deployment functionality into an *agent framework*, which serves as a base for individual agent designs. The framework is intended to be highly generic in nature, to accommodate agents that vary both with regard to supported platform (actual deployment mechanisms depend highly on this) and feature set. That being said, the framework is in reality Linux-specific at this point and will require significant rethinking in order to accommodate other platforms as well.

We've chosen to concentrate on the framework in this discussion as it is responsible for core PyPkg deployment functionality (installation and uninstallation), and should as such provide sufficient insight into the deployment aspect of PyPkg. Our approach in this respect is to provide an alternative take on the information already expressed in great detail in the dedicated design document, included as Appendix B, describing the deployment functionality in terms of distinct *processes*: installation and uninstallation. This approach should help the reader understand the deployment model with regard to offered functionality, as opposed to the development-oriented design document.

Our discussion of the framework design therefore takes place by looking at each process in isolation and how the framework makes it happen. Each process corresponds to a set of high-level demands stated in the design document (Section B.1.2), that serves to define what functionality should be expected from the implemented process. These demands can pertain to both the result of a process and its behaviour during execution. We use the corresponding demands as reference in the treatment of each process.

Since we will be referring to it from this text, it might be useful to first say something about the method we've used to structure the design document. In our design process we tend to think top-down and first of all look at the functionality the system being designed is to include, in terms of demands. Then we can consciously consider how each demand is to be met by system behaviour, in terms of collaborating components. As a system grows more and more complex it becomes only more important to understand the intention behind individual components. Our method lets us develop system behaviour with a specific intention in mind (to meet a demand), and also in hindsight understand the intention behind system structure.

3.3.2.1. The design method

> *We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the*

[1] A reference to an interface version without implementations indicates an inconsistency in the PyPkg network.

[2] The process must be applied recursively for each release added to the dependency chain.

> *program is desirable. But nothing is gained –on the contrary!– by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of.*
>
> *- E.W. Dijkstra [11]*

Central to our design method is the desire to associate external demands with design elements. That is to say, the specified set of demands on PyPkg defines the problem we are to solve, while the design represents the solution, each demand should relate to a subset of the design. What we try to do in our design is to at all times maintain this evasive link between demands and design elements (a demand may affect multiple discrete design elements). The method we have come to rely on for this purpose is to structure the design in terms of *aspects.*

An aspect, with regard to software design, is a filtered view of the system being designed. Working with aspects enables the designer to concentrate on crafting functionality for a single well-defined responsibility (e.g. a demand) at a time, disregarding the complexity of the system as a whole. This line of thought can be traced back to E.W. Dijkstra's assertion that a program is best studied in terms of of separate concerns [11]. Dijkstra uses the words 'concern' and 'aspect' as synonyms, but our use of the latter is inspired by the Aspect-Oriented Software Development (AOSD) paradigm [18]. According to the AOSD definition an aspect is a special kind of concern that crosses module boundaries ("introduces crosscutting in the implementation").

With traditional software development paradigms (object-orientation for instance) concerns have been approached by enclosing them in modules (as per the "separation of concerns" principle), such as classes and functions. Not all concerns map so easily to structural elements however, some affect the design on the global scale (crossing module boundaries). One may for instance consider transaction management in a system that involves database logic. Parts of the PyPkg agent framework need to access a database, and as a consequence database transaction management is encountered in relation to several tasks (not localized within a certain module).

Aspect-orientation augments the object-oriented "toolbox" with a way to explicitly model crosscutting concerns. Concerns that are of a global nature (shattered across modules) are each assigned an aspect, that describes which system components (modules) are affected and how they are affected. As such, one could say that an aspect is a meta-component. An important part of AOSD is techniques for actually implementing crosscutting on the program level, but this requires programming language support for defining aspects (AspectJ being one example [1]).

In conjunction with design of PyPkg we see aspects as generalized concerns, that is, we consistently describe the model in terms of aspects regardless of whether they introduce cross-cutting. Also, we use aspects as completely abstract entities that never result in actual crosscutting in the implementation. If there were aspect-oriented constructs in our implementation language of choice (Python), the situation would likely be different. As it stands, we use aspects simply to describe how our design meets identified responsibilities. Specifically, it enables us to look at a certain responsibility and from the corresponding aspect decide exactly which parts of the design it pertains to.

The set of demands on the PyPkg deployment layer define the top-level responsibilities, so a set of "core" aspects is deduced from these (refer to Section B.1.2). However, many more aspects inevitably surface during development, as aspects are further decomposed in order to

better handle complexity. So aspects are used by us as a general tool for describing functionality present in the framework, on different levels. For simplicity however, we'll concentrate on those aspects of the framework that correspond to stated demands.

3.3.2.2. The installation process

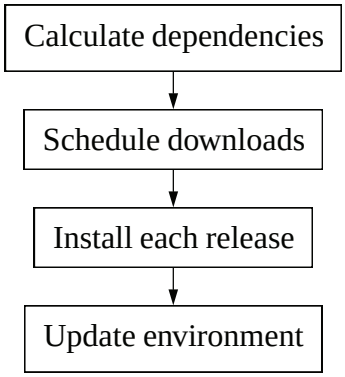

Figure 3.9. PyPkg installation process

With regard to the design document the installation process corresponds to the top-level Installation aspect, in Section B.2. The process has a specific outcome that is easily characterized, but is subject to several demands (hence the various sub-aspects of Installation). The installation process takes as "input" a certain project release that the user wants to install, when the process finishes the software should be completely integrated with the user's system, so that it is in a usable state[1] as intended by its provider. The design document defines the demands that govern the installation process in a list that we repeat here:

1 All unsatisfied dependencies shall be satisfied in the installation process

2 Packages in the following formats shall be supported for installation: PyPkg

3 The installation process shall arrange for background acquisition of packages

4 Installation into both system-wide and user-specific locations shall be supported

5 Installations shall be completed by updating the host environment accordingly

Figure 3.9 illustrates the installation process in a simplified manner that should be relatively easy to grasp. The same information, but in much finer detail and conveyed in a strict technical manner, is presented in the design document's description of the Installation aspect and its sub-aspects (such as Installation.DependencyHandling).

The first step of the process relates to the first item of the list of demands; starting with the set of dependencies for the project release originally scheduled for installation, unsatisfied

[1]Bear in mind that it might be necessary to refresh the environment, i.e. start a new shell, in order to use updated settings. This is something that is outside of PyPkg's reach however.

dependencies are recursively resolved into a queue of corresponding project releases. The queue forms a plan of project releases that are to be installed, ordered so that dependencies come before dependents (meaning that the originally scheduled release comes last).

The next step, corresponding to the third item of the list, is to schedule PyPkg packages for each project release to be downloaded, in a background thread. The PyPkg distribution protocol ensures[1] that each project release has a package associated with it, so the package URL can simply be found from the RDF description for each release to install. The installation thread and the download thread collaborate, so that the installation thread is notified asynchronously by the download thread each time a package finishes downloading.

The third step of the process is where the installation thread installs the package corresponding to each project release in its installation plan. Since packages are downloaded in another thread, it will for each package have to wait until notified unless the package has been downloaded already. Package content consists of two parts, metadata and archived files. The metadata pertains to environment settings and can be used to communicate contained directories that must be added to a certain path, for instance a nested directory containing Python modules that should be part of the Python interpreter's module path. The file archive is organized in a special manner, files are placed in "symbolic directories" according to their type. For instance, executables are placed in "@Executable", Python modules are placed in "@PythonSite" etc. Directories can also be nested within symbolic directories.

Before installation takes place proper, the unpacked package content (files) is rearranged by translating symbolic directory names into "real" filesystem directory names. "@Executable" is translated into "bin" etc. A benefit from operating with symbolic directory names though is that when installing a package into a separate directory, which PyPkg always does at this point, we know which directories to add to which paths (necessary when installing outside of standard paths). For instance; the "@Executable" symbolic directory of a certain package is turned into "<package-path>/bin" when it is installed, we know now that this directory should be added to the `PATH` environment variable so that contained executables are in the shell's path. Nested directories may also be eligible for augmenting corresponding paths, if indicated by package metadata.

When the package has been unpacked and its content properly arranged, it can be installed. Where the installation is directed depends on the configuration of the PyPkg agent, as per demand number four the framework can be configured to install into system-wide or user-specific locations. In order to enable uninstallation the installed release is registered in PyPkg's installation database. Also, as part of installation an internal PyPkg environment representation is updated with settings for the software being installed. These settings correspond to various paths that must be augmented with newly installed directories: `PATH` for directories containing executables, `LD_LIBRARY_PATH` for directories containing shared object libraries etc.

After the whole installation plan has been processed, the last step is to update the host environment, in order to meet the last demand. The framework keeps an internal abstract environment representation, which is during this step propagated into the host environment. The general environment mechanism we use is the shell environment. The user's or the system-wide (depending on the agent configuration) "bash" shell startup file is modified to include a shell script of our own that basically sets a number of variables with corresponding values. The "zsh" shell is

[1] Implementation-wise, this is enforced on the client level.

also supported for systems where it is installed. A special mechanism is used for placing Python modules in the Python interpreter's path.

For Python we employ instead a special `sitecustomize` module which is placed in the interpreter's path with the help of the `PYTHONPATH` shell variable. When the interpreter is started it loads our module, which proceeds to augment the path with directories containing modules from PyPkg installations. This approach gives us much greater control than simply using the `PYTHONPATH` shell variable which is the other alternative available to us.

3.3.2.3. The uninstallation process

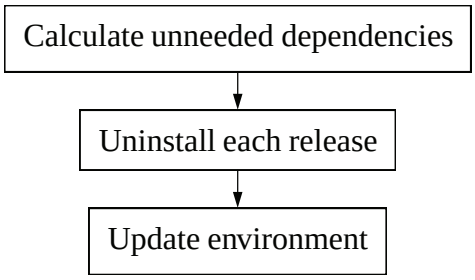

Figure 3.10. PyPkg uninstallation process

The uninstallation process corresponds to the top-level Uninstallation aspect of the design document, Section B.8. This process is much simpler than the installation process, as evidenced by the comparatively low number of demands imposed on it. It can be characterized as taking as "input" an installed project release that the user wants to uninstall, after the process has finished the software should be cleanly removed from the system. Not only should the installation's files be removed, its effect on the system with regard to environment settings and installed dependencies should also be undone. To be concrete, there should be no settings pertaining to the installation left in the host environment and unneeded dependencies of the installation should be uninstalled as well. By unneeded dependencies we mean software that has been installed in order to satisfy dependencies, and that is no longer depended on or desired on its own (PyPkg supports flagging of individual installations as desired on their own, so they are not removed when no longer depended on). The list of demands on the uninstallation process formalizes these concerns:

1 The uninstallation process shall support removal of orphaned dependencies

2 When uninstalled, an installation's effect on the host environment shall be undone

Figure 3.10 illustrates the uninstallation process in simplified terms. A technical description in finer detail can be found in the design document's sections on the Uninstallation aspect and its sub-aspects.

Kind of the inverse of the first step of the installation process, the uninstallation process starts by considering dependencies of the release scheduled for uninstallation (and their dependencies etc.) that are left without dependents (i.e., they are orphaned) in a recursive manner, corresponding to the first demand. An orphaned dependency is left alone if it's marked as desired

on its own, however. Each installation has a special boolean variable that can be set in order to indicate that an installation should not be automatically uninstalled (the user wants to keep it), this is automatically true for software installed on the user's explicit request. Installations that are in this way detected as eligible are queued for uninstallation, dependents before dependencies. Uninstallation is performed in this order to avoid broken dependency relations if the process should be stopped prematurely.

Since there is no need to download anything the next step of the process is straightforward, each queued release is processed in sequence. For each release its files are removed from the filesystem, it is removed from the installation database and its settings are removed from the internal environment representation.

The final step concerns itself with the last demand, i.e. it synchronizes the updated internal PyPkg environment with the host environment. This actually happens in the exact same manner as for the installation process, since this step simply comes down to propagating internal environment settings into the host environment.

3.4. The dependency scheme

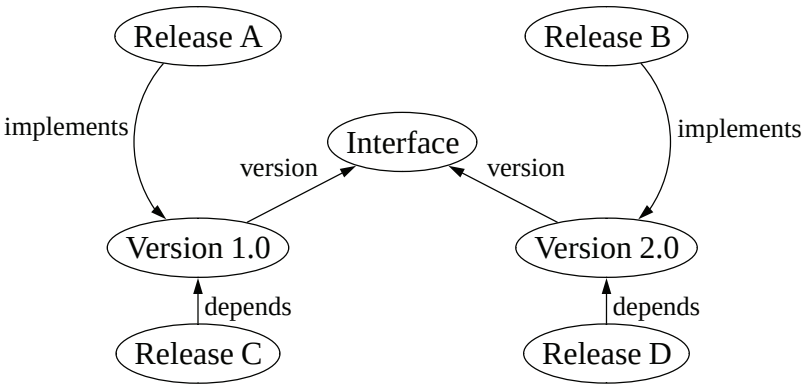

Figure 3.11. The PyPkg dependency scheme

PyPkg uses a special scheme for representing inter-software dependencies, where software requires certain other software to be installed prior to its usage. An example of such a dependency is when a Python program imports modules from a separate library, in order to invoke functionality exposed by those modules. Dependencies are in the PyPkg context expressed on the distribution level, through terms in the DOAP/PyPkg vocabulary (Section 3.2.2.2). The idea behind explicitly stating dependencies as part of distribution metadata is sharing of knowledge: due the general nature of RDF, PyPkg dependency descriptions can for instance be translated into a presentation format (i.e. HTML) for informing users of requirements when distributing outside of PyPkg.

Central to the PyPkg dependency scheme is the *interface* concept. An interface in this regard is an abstract notion, a way of formalizing properties of software components that other software may depend on. Since such a dependency in practice inherently comes down to a reliance on properties of external software components it is only natural to formalize this relation. That is,

rather than relying on tacit knowledge of which components satisfy a dependency it is better to state a contract for what is being depended on, so that components adhering to this contract can be used to satisfy the dependency. PyPkg represents such contracts in terms of interfaces. Furthermore, since software components tend to change their outward interface from time to time, interfaces in the PyPkg dependency scheme are *versioned*.

Given that dependencies in PyPkg are stated in terms of interface versions (Section 3.2.2.2 describes this procedure), which are abstract entities, there must be a way of resolving them into concrete deployable releases. The dependency scheme solves this through the notion of *implementations*; when a project makes a release, the release is also declared as implementing a certain interface version. When this release is installed, it will expose the interface version in question, and as such any corresponding dependency is satisfied as well. Thus, dependencies are according to our dependency scheme resolved through knowledge of implementations.

The dependency scheme is exemplified by Figure 3.11, where releases C and D express dependencies toward certain interface versions. Also, releases A and B state that they implement versions 1.0 and 2.0 of the interface, respectively.

In the following subsections we first explain the rationale for representing dependencies in software with the help of interfaces, in comparison to conventional dependency management, before we describe in depth the semantics of the PyPkg interface concept.

3.4.1. Rationale

Automatically resolving dependencies when installing software is quite an involved task, especially when considering that a dependency in the general case can resolve to several different software products. For the most part this will come down to different versions of the same product, but sometimes independently developed software may deliver converging functionality (e.g., a dependency may be met by two different libraries). When it comes to choosing between versions for resolving a dependency, one needs to know *which* versions are compatible. Software components tend to break compatibility from time to time as they evolve.

The APT model for package management offers integrated support for expressing and handling (upon deployment) dependencies. The APT-derived Portage package management system, which is the most familiar to us, expresses dependencies toward other packages as part of package metadata. Such specifications happen in terms of package identifiers and sets of compatible versions [22]. Support has also been added for specifying abstracted package identifiers, so-called virtual targets. The gist of virtual targets is that they may be satisfied by several concrete packages, which declare that they correspond to the target in question.

We consider this conventional dependency scheme rather weak and inflexible. Instead we contend that a formal scheme of this sort should precisely reflect the actual nature of inter-software dependencies. What one piece of software requires from another, in this respect, is a type of service. The dependent will invoke the interface exposed by the dependency, in order to achieve certain functionality. As such, it is really the interface that is being depended on. Which concrete software, or versions of which, implements the interface is less relevant. What matters is that an implementation corresponds to the contract represented by the interface. Which implementation takes precedence can eventually be controlled through rules (no such functionality in PyPkg, yet).

Therefore, PyPkg formalizes software dependencies in terms of abstract *interfaces*, where

an interface represents functionality that a piece of software exposes as a service to other software. Since other software may come to depend on this interface, it must be considered to adhere to a contract with regard to its constituent parts and their semantics. Thus, when software states dependencies on interfaces it should be unambiguously clear what is being required. In order to satisfy dependencies expressed in this manner, abstract dependency specifications must be resolved into concrete software releases that implement (expose) the corresponding interfaces.

3.4.2. Semantics of the interface concept

Certain semantics are defined for the PyPkg interface concept, which affect the PyPkg system in various ways. The semantics are documented here in conjunction with different aspects of the dependency scheme.

3.4.2.1. Creating interfaces

An interface comes into existence automatically when a project is introduced to the PyPkg distribution network, since a project is considered to expose an interface through its software (Section 3.2.2.2). Therefore an interface simply exists as an identifier, the URI for the corresponding project. There are currently no special terms in the DOAP/PyPkg vocabulary for describing an interface.

Even though we use interfaces to model contracts for software components, it should be said that we don't formalize the actual properties represented by such contracts. This must instead be communicated through other means by the creator of the interface, so that others know what functionality to expect if they are to depend on it. We have considered adding terms to the DOAP/PyPkg RDF vocabulary so that providers may, through free text, document interfaces, but this has not been done yet. It is definitely worth considering as a future improvement though.

3.4.2.2. Interface versions

Interface versions are inherently linked to project releases, a specific version comes into existence the first time it is associated with (implemented by) a release.

Interface version numbers consist of two components, the major and minor version: <major>.<minor>, e.g. 2.0. Each time an interface is augmented with new properties (backward-compatible changes) the minor version gets incremented, when backward-incompatible changes occur the major version gets incremented and the minor version is set to zero. The gist of this formal versioning scheme is that interface versions with different major components are incompatible with each other, while those with equal major components maintain backward compatibility as the minor version increases. So that when for instance a dependency is stated on version 1.2 of an interface, it is possible to satisfy that dependency with software that corresponds to version 1.4 of the same interface.

3.4.2.3. Interface implementations

The owner of an interface, a project in the PyPkg distribution network, is required to provide implementations for all versions of the interface. Thus, when a deployment agent encounters a dependency statement it can look up the corresponding project and find a compatible implemen-

tation. Specifically it will have to find a project release that implements a version of the interface with the same major version as the one depended on, and with a greater or equal minor version.

Chapter 4. Usage

This chapter goes into practical usage of the PyPkg software distribution/deployment system, as it manifests itself both to software providers and end users. The reader may recall from the discussion of the product vision that PyPkg is conceived as a kind of conduit for transferring software (Section 2.2.1). In this chapter we intend to show how this notion has in fact been realized. We go about this by studying how a specific project is both distributed (provider side) and deployed (user side) through PyPkg. For this purpose we chose SciPy, a subset of the Simula Suite software collection described in Section 1.1. SciPy is a fairly complex example, but all the same far less involved than the full suite. As such it should prove a suitable example.

Two tools have been developed to facilitate the different usage aspects of PyPkg so far. As providers only need to present themselves statically through DOAP/PyPkg descriptions in RDF/XML, they can get by without extensive tool support (although manual XML editing can be a chore). The exception in this respect is package authoring, a task which is decidedly best automated (given that PyPkg packages involve metadata and must follow a certain structure). Therefore, although the end user perspective has had priority during development, we've developed a simple graphical package authoring tool: PyPkg.PkgMaker. In support of the end user perspective, we've developed a graphical, "wizard"-like, tool that implements installation through PyPkg: PyPkg.Installer.

The PyPkg agent framework does include support for uninstallation of software, but we found it right to concentrate development efforts on first implementing a functioning installation tool, before spending time on tool support for installation management.

We are going to explain the process of transferring SciPy software from one party (the provider) to the other (the end user) with the help of the PyPkg system, by looking at how it presents itself to each. Thus, the chapter is divided into two main sections; the first showing how SciPy is introduced to the PyPkg network, the second showing how it is installed by the end user.

4.1. Distributing SciPy

This section is concerned with detailing the process of publishing the SciPy project, and its software, through PyPkg. This is done through an example which involves all steps from creating the DOAP/PyPkg description (an RDF/XML document) to publishing a new SciPy version complete with a PyPkg package and creating an installation script. The example will also include dealing with some other projects, due to dependencies of the SciPy software.

4.1.1. Publishing the project

In order to introduce SciPy to the PyPkg network, and by extension PyPkg users, we must first create a DOAP/PyPkg document for it. This document shall describe the project to parties (agents) familiar with the vocabulary. To this end, we start off with a basic description of the

project itself using only terms from the DOAP vocabulary. A generic (somewhat simplified at that) DOAP description of the SciPy project is shown in Figure 4.1. For instance it states that Eric Jones is one of SciPy's maintainers and Prabhu Ramachandran is one of the developers (there are more people associated with the project, but we left them out to conserve space). For a comprehensive explanation of the DOAP vocabulary see [16].

With the general DOAP description of SciPy down we can begin adding support for the PyPkg deployment aspect. Concretely, we must add a DOAP resource representing the latest SciPy release (using `doap:Version`), and declare dependencies toward other software with the help of PyPkg extensions (i.e. `sw:dependency` etc.). Figure 4.2 shows the RDF/XML description of the latest SciPy release, complete with PyPkg-style dependency declarations. All dependencies refer to other projects accessed through the central PyPkg portal (currently located at http://kangchenjunga.simula.no/arvenk/doap). Each of these projects must declare at least one release that implements the interface version being depended on (for instance 2.4 for simula.no/python), otherwise the dependency is invalid (an interface version doesn't come into existence on its own). One can see from the description that SciPy version 0.3.2 implements revision 0.3 of the scipy.org/scipy interface (the interface is implicitly taken to be the project itself), by consulting the `sw:implements` property.

The SciPy DOAP/PyPkg description is now in such a state that it can be published. Currently the PyPkg portal system is very primitive: projects must be located directly at a central server (the same as the portal) and the portal is implemented directly by hosting an RDF/XML document listing all providers. The portal operates with a namespace where all projects must be associated with a provider, for SciPy we decide on the provider identifier "scipy.org". Unique project identifiers are obtained by combining the provider's identifier with a provider-relative project identifier (joining the two by a slash). The resulting PyPkg identifier for SciPy is "scipy.org/scipy".

Accordingly the project document is placed at <portal-uri>/scipy.org/scipy[1], so that the project can be reached by combining the project identifier with the portal's base URI. The project is not yet visible through the portal's project index however (which is how agents discover available projects). Thus, the last step is to register scipy.org as a new provider with the portal index (http://kangchenjunga.simula.no/arvenk/portal.rdf). The portal uses a special RDF vocabulary, `swidx` for "software index", for describing the individual provider. The description of scipy.org (and providers of scipy.org/scipy's dependencies) can be seen in Figure 4.3. Notably, the class `swidx:Provider` describes a provider, and property `swidx:project` associates the provider with a project (as one can see scipy.org also provides Numeric).

4.1.2. Adding a PyPkg package

So far, a release (version 0.3.2) has been added to the SciPy description. In order for it to be accepted by PyPkg deployment agents it must be associated with a PyPkg package, though. This comprises the creation of a package in the PyPkg format, and describing it as a property of the SciPy 0.3.2 release. Specifically, PyPkg agents expect to see that the version (`doap:Version`) in question has an `sw:package` property, referring to a package resource of the `pypkg:Package` type.

[1] <portal-uri> meaning http://kangchenjunga.simula.no/arvenk/doap.

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
   xmlns='http://usefulinc.com/ns/doap#'
   xmlns:foaf='http://xmlns.com/foaf/0.1/'
   xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:sw='http://kangchenjunga.simula.no/arvenk/ns/sw#'
   xmlns:pypkg='http://kangchenjunga.simula.no/arvenk/ns/pypkg#'
   >
   <Project>
      <os>OS portable</os>
      <maintainer>
         <foaf:Person>
            <foaf:name>Eric Jones</foaf:name>
         </foaf:Person>
      </maintainer>
      <developer>
        <foaf:Person>
          <foaf:name>Prabhu Ramachandran</foaf:name>
        </foaf:Person>
      </developer>
      <programming-language>Python</programming-language>
      <description xml:lang="en">SciPy is a collection of mathematical
algorithms and convenience functions built on the Numeric extension for Python.
It adds significant power to the interactive Python session by exposing the user
to high-level commands and classes for the manipulation and visualization of
data. With SciPy, an interactive Python session becomes a data-processing and
system-prototyping environment rivaling sytems such as Matlab, IDL, Octave,
R-Lab, and SciLab.
      </description>
      <created>2001-05-21</created>
      <license rdf:resource="http://usefulinc.com/doap/licenses/bsd"/>
      <homepage rdf:resource="http://www.scipy.org"/>
      <download-page rdf:resource="http://www.scipy.org/download/"/>
      <shortname>scipy</shortname>
      <shortdesc xml:lang="en">An open source library of scientific tools for
Python, supplementing the Numeric module.</shortdesc>
      <name>SciPy</name>
   </Project>
</rdf:RDF>
```

Figure 4.1. SciPy DOAP/PyPkg description

The PyPkg package format is relatively involved, in that it encompasses certain metadata and prescribes a certain organizational structure of contained files (see Section B.4). Therefore we always use an automated tool for building packages: PyPkg.PkgMaker. Based on information

```
<release>
  <Version>
    <revision>0.3.2</revision>
    <created>2004-10-13</created>
    <sw:implements>
      <sw:Implementation>
        <revision>0.3</revision>
      </sw:Implementation>
    </sw:implements>
    <sw:dependency>
      <sw:Implementation>
        <sw:interface rdf:resource=
"http://kangchenjunga.simula.no/arvenk/doap/simula.no/python"/>
        <revision>2.4</revision>
      </sw:Implementation>
    </sw:dependency>
    <sw:dependency>
      <sw:Implementation>
        <sw:interface rdf:resource=
"http://kangchenjunga.simula.no/arvenk/doap/scipy.org/numeric"/>
        <revision>24.0</revision>
      </sw:Implementation>
    </sw:dependency>
    <sw:dependency>
      <sw:Implementation>
        <sw:interface rdf:resource=
"http://kangchenjunga.simula.no/arvenk/doap/cens.ioc.ee/f2py"/>
        <revision>2.45</revision>
      </sw:Implementation>
    </sw:dependency>
  </Version>
</release>
```

Figure 4.2. Description of SciPy version 0.3.2

from the package author, the tool will even upload the built package to a specified location and add an appropriate description to the corresponding DOAP/PyPkg document.

When started, PyPkg.PkgMaker presents a graphical form for gathering necessary information from the package author. This information can be grouped into two categories. The first pertains to transforming a source package into the (binary) PyPkg format, which comes down to fetching the corresponding archive from a certain location and building it according to certain rules, before the result is packaged. The second category pertains to the package's publishing process: uploading it to a certain location and updating the associated project's description.

The build rules employed depend on the build method of the source package, PyP-

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:swidx="http://kangchenjunga.simula.no/arvenk/ns/swidx#"
   xmlns:doap="http://usefulinc.com/ns/doap#"
   >
   <swidx:Provider>
      <doap:name>SciPy</doap:name>
      <swidx:id>scipy.org</swidx:id>
      <swidx:project>scipy</swidx:project>
      <swidx:project>numeric</swidx:project>
   </swidx:Provider>

   <swidx:Provider>
      <doap:name>cens.ioc.ee</doap:name>
      <swidx:id>cens.ioc.ee</swidx:id>
      <swidx:project>f2py</swidx:project>
   </swidx:Provider>

   <swidx:Provider>
      <doap:name>Simula Research Laboratory</doap:name>
      <swidx:id>simula.no</swidx:id>
      <swidx:project>python</swidx:project>
   </swidx:Provider>
</rdf:RDF>
```

Figure 4.3. Portal index

kg.PkgMaker includes support for several such methods: GNU (Autoconf [27]/Make), Boost Jam (for the C++ Boost library) [38] and Distutils (standard for Python software). If a source package doesn't quite fit with any of the predefined build methods, the build process can be scripted, partially or completely. The latter option is chosen by selecting "Custom" out of the tool's supported build methods. The scripting itself takes place by supplying up to three Python methods corresponding to steps in the build process (if the "Custom" method is chosen, all steps must be supplied): `configure`, `build` and `install`. The first step is dedicated to any preparation of the source package before building, the second should take care of building a binary representation from the package source, while the third is required to install all files to be packaged into a temporary directory. Functions from the predefined build methods (Distutils etc.) are exposed to build scripts, so slight customizations of the default build rules can be accomplished with relative ease.

Additionally, if the package contains separate directories which content should be exported via the environment upon installation (e.g. nested directories containing shared object libraries), this is possible via a section in the GUI labeled "Exports". This section contains fields for different types of files (executables, object libraries and Python modules) where package directories can be filled in. The various directories are communicated through metadata elements in the re-

sulting package.

Figure 4.6 shows an invocation of the PyPkg.PkgMaker tool, where the form has been filled in in order to build a package for SciPy version 0.3.2. As we can see from the chosen Build method, SciPy uses on the source level the Distutils framework, albeit with certain modifications. Since SciPy deviates somewhat from the normal Distutils method, we have to adapt the build process a little bit. Support for the SciPy Distutils variant was added to the PyPkg.PkgMaker script API, so all that is needed is an invocation of the correct method (`_distutils_build`) for the "build" step. The build recipe (script) for the SciPy package is shown in Figure 4.4. The recipe also shows that the Distutils build engine must be set up to "include" the Numeric library, a dependency of SciPy.

```
def build(self):
    self._distutils_build(requiredLibs=["scipy.org_numeric-24"], scipy=True)
```

Figure 4.4. SciPy build recipe

Figure 4.5 shows the resulting DOAP/PyPkg package description (in RDF/XML syntax) that is added by PyPkg.PkgMaker to the SciPy project document. A nice feature of automatically generating such descriptions is that size and checksum properties are calculated for you.

```
<sw:package>
  <pypkg:Package>
    <sw:download-url>http://kangchenjunga.simula.no/arvenk/pkgs/scipy-0.3.2.pypkg
    </sw:download-url>
    <sw:md5>e8d0de4d10c41c76a4c934f2073f110a</sw:md5>
    <sw:size>2981603</sw:size>
  </pypkg:Package>
</sw:package>
```

Figure 4.5. SciPy PyPkg package description

4.1.3. Creating the installation script

To aid in installing SciPy we create an installation script, of the PyPkg.Installer type. PyPkg.Installer is a generic PyPkg agent for installing software, that serves as a template for creating product-specific installation scripts. That is to say, PyPkg.Installer is a complete application except that it must be configured to install a certain product (SciPy in this case). The version to install will be determined automatically by querying the corresponding project upon script execution. The resulting script can then be offered to users interested in installing the software in question. A more thorough description of PyPkg.Installer can be found in the next section, which deals with its usage (from the user point of view).

The process for creating product-specific installation scripts from the base PyPkg.Installer template is rather underdeveloped, at this time it is implemented by copying a generic Python

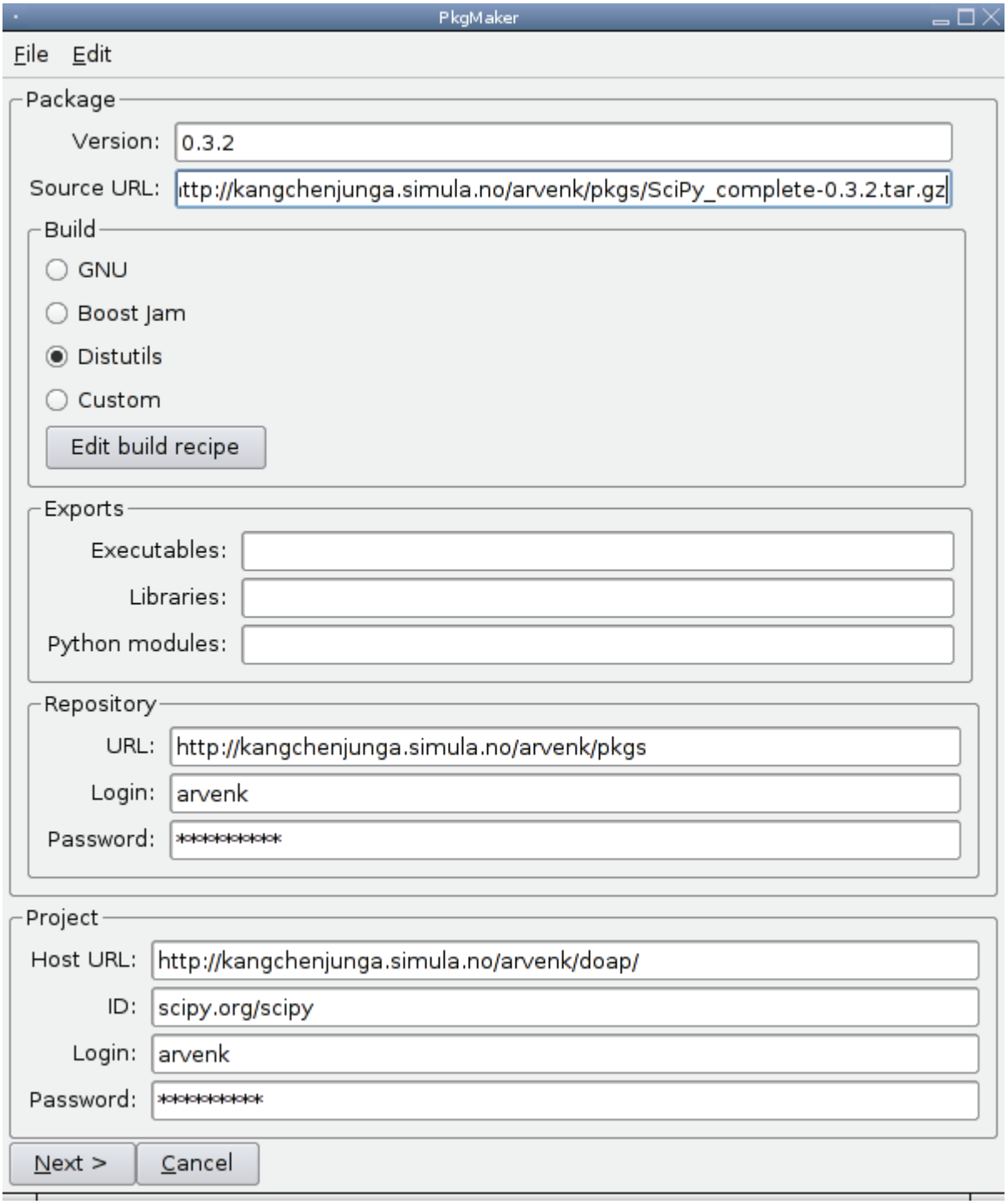

Figure 4.6. Creating SciPy package with PyPkg.PkgMaker

script (`installer.py`) and editing the new script (`scipy-installer.py`) to refer to the product in question, e.g. SciPy. The product to install is specified by defining a class `_Project` with attributes `name` (the project name) and `id` (the PyPkg identifier for the project). Figure 4.7 shows the `_Project` definition for the SciPy PyPkg.Installer script.

```
class _Project:
    name = "SciPy"
    id = "scipy.org/scipy"
```

Figure 4.7. Product data for SciPy installation script

4.2. Installing SciPy

In this section we will show how PyPkg.Installer is used to install SciPy on Linux, as a regular user without administrative privileges. The process takes place by acquiring a PyPkg.Installer script, a single executable (Python script), for SciPy and running it from start to finish. As a result F2PY version 2.45.241_1926, Numeric version 24.1 and SciPy version 0.3.2 are installed beneath the directory .pypkg/root in the user's home directory (the first two are installed to satisfy dependencies of SciPy). The F2PY executable `f2py` is placed in the shell's path (`PATH`) and Python modules installed by F2PY, Numeric and SciPy are added to the Python interpreter's module search path. In order for the new installations to be functional, the user may have to start a new shell.

PyPkg.Installer, at heart, is a type of PyPkg deployment agent, specialized toward installing software. It is designed as a straightforward graphical installation client, guiding the user through the installation process in a manner similar to Windows Installer scripts. The requirements imposed on the user's system are quite modest, PyPkg.Installer needs Python 1.5.2 or newer and the X Window System. Thanks to integrated bootstrapping logic, it will handle other requirements by itself upon execution.

4.2.1. Starting the installation

Once the SciPy PyPkg.Installer script, `scipy-installer.py`, is at hand it can be run in a terminal from within the X Window System. When the script is started it is possible that it will produce messages in the terminal window; if it must acquire a compatible Python version as part of the bootstrapping process, progress will be communicated via a character-based interface as Python is acquired/installed. In order to monitor such messages we recommend that PyPkg.Installer scripts are started from a terminal, even though they are graphical applications (X Window clients). Once `scipy-installer.py` has prepared itself it will greet the user by displaying information about the software to install, as shown in Figure 4.8.

4.2.2. Calculating dependencies

The next step in the installation process, calculating dependencies to install, is invoked by pressing the "Next" button. This step consists of recursively checking which dependencies must be satisfied, and resolving them into installable project releases. Unfortunately PyPkg will currently only check dependencies against its database, so if e.g. Numeric has been installed in the host system by other means it may still be scheduled for installation.

Figure 4.9 shows the result of the dependency calculation for SciPy 0.3.2. As we can see, dependencies cens.ioc.ee/f2py-2.45 and scipy.org/numeric-24.0 have not been met (Python, on

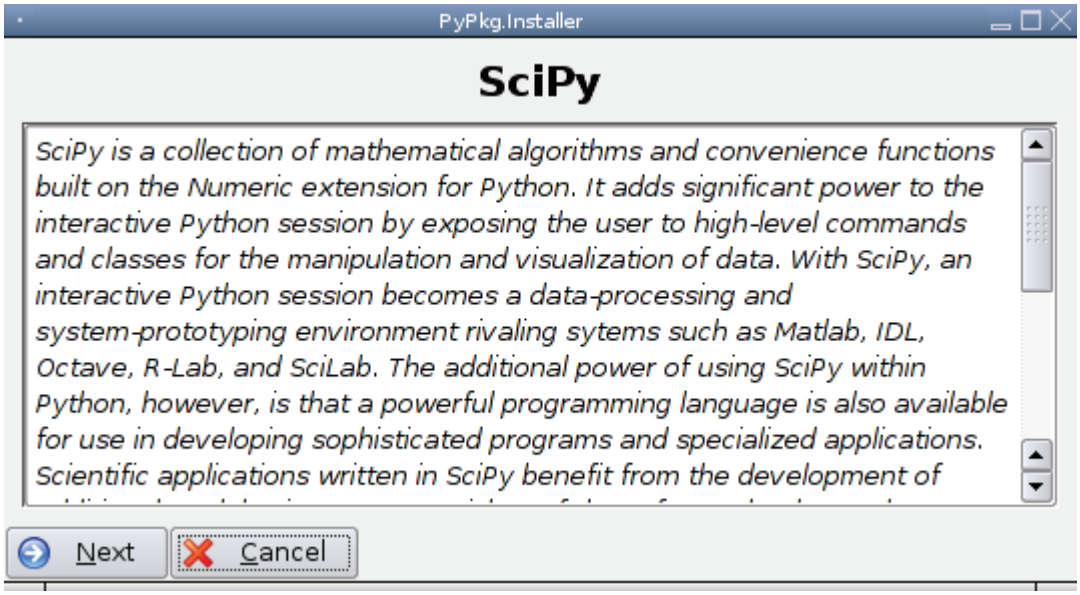

Figure 4.8. `scipy-installer.py` presents itself

the other hand, is installed as a dependency of PyPkg), so they must be satisfied by installing implementing project releases: scipy.org/numeric-24.1 and cens.ioc.ee/f2py-2.45.241_1926. The installation script will also await confirmation, from the user, after this step before proceeding to the next step.

4.2.3. Acquiring and installing packages

The last step of installing SciPy is the installation of packages scheduled during the previous step. This step actually consists of two tasks, downloading packages and installing them, that occur in parallel. That is to say, while a package is being installed any remaining packages are scheduled for download in the background. As such, the GUI for monitoring this step is divided into three sections. Illustrating this, Figure 4.10 shows how SciPy 0.3.2 is being installed. To the left is a list displaying the installation queue in descending order, with an arrow pointing to the package currently being installed (SciPy itself, in the screenshot). At the top is a section that monitors the download process for each package. The progress bar near the bottom shows completion percentage for the package currently being processed; first for the unpacking of the package, then the installation of contained files.

4.2.4. Finishing the installation

After the `scipy-installer.py` script finishes, SciPy should be correctly installed with all dependencies in place. Before making use of it however, i.e. importing it from Python scripts, one should start a new shell or perhaps update the current shell's environment by evaluating its startup file (with bash: "source ~/.bashrc"). This is because PyPkg modifies shell environment variables, in order to make installed files known in the system. For instance, the `PYTHONPATH` variable is modified to include a PyPkg-owned directory which contains a file `sitecustomize.py`. This file is executed by the Python interpreter and adds locations of installed Python

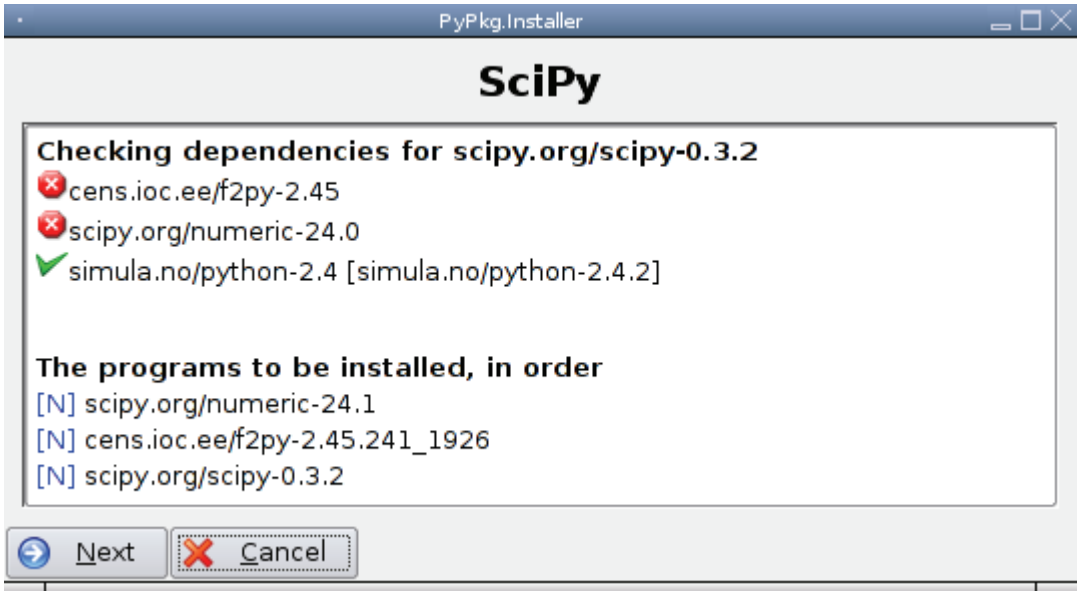

Figure 4.9. scipy-installer.py after calculating dependencies

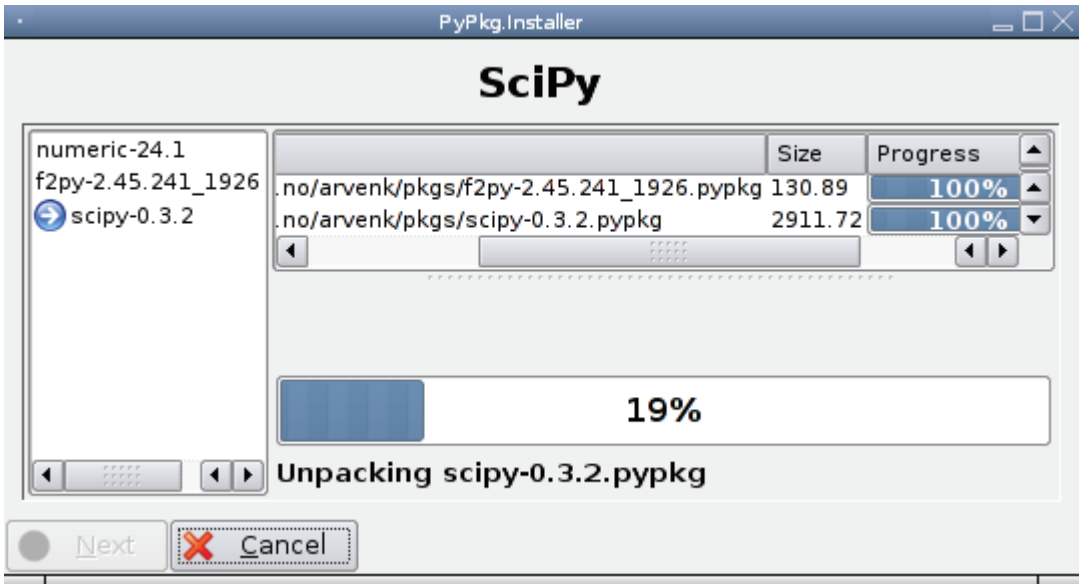

Figure 4.10. Installing SciPy and dependencies

modules to the module search path. In this case, modules belonging to F2PY, Numeric and SciPy are added. When importing the `scipy` module for instance, it is able to import its dependencies (`f2py2e` and `Numeric`) since they are part of the module path.

Chapter 5. Conclusion - Future Directions

Concluding the thesis, we will reflect on the product resulting from our own process. Our approach in this respect is to consider how the PyPkg system corresponds to what we set out to do, from a usability perspective. After all, the ultimate goal of this project is to deliver a product with certain qualities. PyPkg is far from finished, yet work on the thesis ends here and we evaluate therefore PyPkg as it manifests itself at the time of writing. Since this is to be the conclusion to our work, we present also some thoughts on how we think our system has worked out as a solution to the general distribution/deployment problem.

Our intention is for development on PyPkg to continue after this thesis ends, in order to deliver a solid feature-complete system distribution/deployment system. To facilitate further work we provide a second section in this chapter, dedicated to an open-ended discussion on issues we think that the next iteration of PyPkg should address, so that the system can evolve to truly realize our vision.

5.1. Evaluation

According to the vision for PyPkg, outlined in Chapter 2, it is to resemble a conduit for transferring software, as a way to facilitate distribution and deployment in general. So how does the realized system correspond to this ideal? We contend that PyPkg fares rather well, although it has only been implemented on a limited scale so far (in comparison to the full vision). At the time of writing fourteen products have been published through our system (only within Simula), predominantly Python-based. The important thing is that the system manages to establish an unbroken conduit for published software, so that it can be deployed with ease.

5.1.1. Deployment

PyPkg deployment functionality is currently synonymous with PyPkg.Installer, which is quite functional at this point; when a PyPkg.Installer-based installation script is run by the user, the corresponding software is perfectly (as far as we can tell) integrated with the system. To see how SciPy is installed per this approach, refer to Chapter 4, Usage. It is also encouraging that we have not yet encountered any incompatibilities with any of the Linux systems we have installed software on so far: Debian and Gentoo. That being said, PyPkg.Installer is still in need of significant testing and polishing before it can be considered production quality.

A testament to the streamlined operation of PyPkg.Installer is the time it requires to install SciPy. As the reader might recall from Chapter 4, SciPy is a rather big package in itself and also presents some complex requirements in terms of other software components (Numeric, BLAS/LAPACK, F2PY). Installing it and its dependencies from source can literally take hours. When installing SciPy and other required packages (Numeric and F2PY) through PyPkg.Installer on the author's machine[1] however, the whole process takes about 20 seconds, including time spent downloading packages (about 3.4MB in total). It helps noticeably that packages are down-

loaded in parallel to installation, especially since the last package to be installed, SciPy, is also the largest.

We are also quite pleased with the support for per-user installations. This has proved extremely useful in our own work, in a multi-user Linux environment with regular privileges. In addition, any possible havoc resulting from an experimental deployment system will be isolated to a single user account. We imagine that this kind of flexible deployment will find appreciation among prospective users, when PyPkg is eventually released to a wider audience.

A tool for managing installations is missed though. Since there is no clean way of removing installations, we tend to remove the whole PyPkg database in order to "start with a clean slate". This is not a huge problem at this stage, but definitely something that should be remedied soon. Support for automated updates is something we'd like to see as well, especially if we are to deploy software with PyPkg on a regular basis.

5.1.2. Distribution

The user side of PyPkg (deployment) is the most important to us, especially since there's little reason to open the distribution network for independent providers until the deployment layer provides the desired base functionality. As a result support for publishing software through PyPkg is a little on the slim side; package authoring is handled by a comprehensive tool (PyPkg.PkgMaker), but project representations in RDF/XML must be edited by hand.

Convenience aside, the approach of writing DOAP/PyPkg descriptions of projects responsible for each distributed product appears perfectly sensible; it allows us to state all relevant information in one place, both that pertaining to humans (project metadata, such as developer information) and that enabling automated deployment. If the format turns out to be inadequate at some point, it can be extended (as we originally extended DOAP). The description format being completely generic, it can be mined directly for any conceivable purpose. We definitely prefer this way of working compared to having to deal with individual software repositories (Sourceforge, Freshmeat) and/or packaging software for mutually exclusive package management systems (Portage, APT).

5.1.3. Summing up

In our view PyPkg very much makes good on its promise of delivering an unbroken conduit for connecting users with providers of software, particularly from the user point of view. The system should still be considered on the prototype stage; the installation tool (PyPkg.Installer) needs further testing/stabilizing, but it is nearing a level of maturity where it can be released as part of a pilot test (in conjunction with the aforementioned Simula Suite).

Also, as we've applied PyPkg to practical problems, such as installing SciPy via PyPkg.Installer, our confidence in the underlying idea has only grown. Both when it comes to deploying software and distributing (providing) it, we have a strong sense that we're moving in the right direction with the project, and that the end result is something we will want to use over current alternatives.

---

[1] A rather mediocre desktop PC, notable hardware includes an AMD XP2400+ CPU and a Hitachi 7200 RPM hard drive, the Internet connection is 3500Kb/s ADSL.

5.2. Future directions

PyPkg is no doubt work in progress, existing functionality needs further attention to attain the required maturity and the system must also be extended in order to realize the vision's full extent. The number one priority for us is to fully implement the current, functionality-wise limited, model (Chapter 3) before starting to add new features we'd like to see in PyPkg. That is, there should be proper functioning support for installation and uninstallation of software through PyPkg, so that we have a quality base system to improve on. This goal needn't be too far away, considering that PyPkg already has relatively good installation support in PyPkg.Installer and that preliminary uninstallation support exists within the deployment framework.

In any case it's interesting at least for the sake of this thesis to document ideas for how PyPkg may be extended in the future, with regard to keep improving the product. Especially since it presents such limited functionality at this stage, any number of things may come to mind in this respect. In order to be effective about it however, and ensure timely progress, it is pertinent to sift out a certain subset, similar to how the current model operates with a restricted scope.

There are therefore two aspects to our discussion of future directions for PyPkg development: what needs to be done to reach a certain maturity, and what issues should be taken on when developing the next version of PyPkg.

5.2.1. Solidifying base functionality

There are some concrete issues that should be solved before the base PyPkg system (i.e. the initial version) represents a satisfactory solution, to a degree that we are confident about introducing it to the public. We articulate therefore a plan for achieving the desired level of maturity.

Deliver functioning PyPkg.Installer for Simula Suite

The primary goal for the PyPkg project at the current stage is to deliver the Simula Suite software package (see Section 1.1) with functioning deployment support, so that it may be released as part of a limited pilot test. It is important for us to solidify implemented deployment functionality, and the best way to go about this is to apply it to practical problems. For the early testing phase it should be enough to deliver a working PyPkg.Installer script for installing the Simula Suite on the systems of testers. It is impractical not to have any equivalent tool for undoing installations, but PyPkg installs all software into separate directories (for system-wide installations beneath /opt and for user-specific installations beneath ~/.pypkg/root) which at least facilitates manual uninstallation[1].

Uninstallation support

Once PyPkg.Installer is relatively mature the next goal should be to deliver a complementary uninstallation tool. To be specific, this tool should be aimed at general management of installed software but initially restricted to uninstallation support. When this tool is sufficiently tested, it should in combination with PyPkg.Installer represent a viable alternative for deploying software on Linux systems.

---

[1] If an installation is removed from the filesystem, it will still be registered in the PyPkg database. Due to the lack of an uninstallation tool, the only option for clearing such data is to delete the whole database.

Facilitate distribution

Supporting the user point of view is the most important to us, especially since we handle all software distribution through PyPkg during the initial phase ourselves. Once we feel confident about presenting PyPkg to the general public though, it is time to consider support for independent software providers. In order to at all enable third party projects to enter the system, a proper indirection system for the portal is needed, so that projects located elsewhere can be registered. With the current ad-hoc portal, projects must reside directly on the portal server.

Additionally, distribution-related tasks, such as creating RDF project representations, should be facilitated, especially in order to ease the transition for independent software providers looking to join the PyPkg network. Currently we see that there are two aspects to this: registering projects with the central portal and manipulating project representations. We see creation/publishing of packages as part of the latter. Concretely, we foresee that a Web interface is created to allow providers to register themselves with the PyPkg portal and update their representation (add projects), and that a special tool is developed with support for all tasks related to creating and maintaining project descriptions. The latter would include the functionality that is now offered by PyPkg.PkgMaker, i.e. support for authoring packages and adding them as part of release announcements.

5.2.2. Extending the PyPkg system

The purpose of this subsection is to identify shortcomings of the currently defined model with respect to the product vision. As such, we are now free to consider new goals for upcoming development iterations. Still, it is important to practice restraint in this process. There are many potential ways in which the system may be improved, but we must keep in mind that they must be realizable within a reasonable timeframe. With this in mind we've decided on a number of distinct concerns on which to base development of a future version of PyPkg.

The individual concerns are ordered according to priority, each pertaining to an idea for how PyPkg may be improved. Even though we've applied a certain ordering, it's not to say that different concerns can't be handled in parallel. After all, part of the rationale for identifying separate concerns, when engineering a system, is that they can be assigned to different teams.

Multi-platform deployment

The envisioned software conduit is intended to present a completely general method, not restricted to any particular platform. The initial version of PyPkg is a Linux-only solution, but we see no particular reason for it to stay this way, it should definitely be possible to achieve a clear separation between generic and platform-specific logic within the deployment model. The way we see it the model should employ a layered architecture, where platform-specific logic resides on the bottom. The first step toward a functioning model of this sort would be to clean up the framework and move the Linux-specific logic into a layer of its own, once this works as expected we can start considering support for other platforms.

Another aspect of multi-platform support is that of discerning between software packaged for different platforms. That is, with the current, Linux-only, system we can assume that PyPkg packages are for Linux (on i686[1] hardware to be exact). If we are to support other platforms as well, this assumption will no longer hold. What must specifically be done is to extend the

DOAP/PyPkg model so that we can describe how packages relate to deployment platforms.

Introducing the notion of platform-dependent packages carries with it some implications that must be considered as well; the exact set of dependencies may vary from package to package, and when checking whether there exist any implementing packages for a dependency it must be relative to platform. It may be a good idea to allow dependencies to be stated both per release (applicable to all associated packages) and per package.

With sufficient support in both the distribution and deployment layer of PyPkg, it should be possible to start implementing deployment support for another platform than Linux. Windows is a likely choice in this regard, given the amount of users on this platform, Mac OS X being another contender. What will happen in terms of development for other platforms than Linux depends ultimately on the expertise of developers eventually joining the project.

Support for on-site building of packages

Per special request from scientists at Simula we are considering the addition of support for building packages from source, on the user's system, prior to installation. This request came about due to the popularity of the ATLAS BLAS/LAPACK library in the scientific computing community. ATLAS is but one of many BLAS/LAPACK implementations, its claim to fame being performance. ATLAS achieves its level of performance by deducing optimal build parameters for each machine (taking hardware properties such as CPU cache size into account), so (presumably) it's not very well suited for distribution in generic binary packages.

The Simula scientists believe that the acceptance of ATLAS-based PyPkg packages for scientific computing purposes, which is of particular interest to Simula, may depend on per-system-optimized ATLAS binaries. The average scientist may simply perceive generic ATLAS binaries as inferior to a build optimized for his/her system, regardless of any actual performance numbers.

Exactly how we should go about accommodating specially built packages is for the moment unclear. It is possible that the most sensible approach would be to allow some sort of local PyPkg portal that lets users substitute their own packages for the generic ones, this approach would at least not affect the installation process itself.

Portal Web interface

O'Reilly CodeZoo is a shining example of how freestanding DOAP metadata for software projects can be aggregated into a comprehensive software directory, so that users can easily find what they are looking for. When adding a DOAP-described project to CodeZoo, the data are read/utilized directly and updates to the project description (such as new releases) can also be picked up directly through a so-called "DOAP-over-Atom" feed [30].

Software is arranged both according to programming language and categories which makes for very convenient browsing of the directory, in addition it is possible to search the project database. We find this solution very appealing, since it manages to combine homogeneous presentation of software with ease of maintenance on the providers' part (thanks to DOAP-over-Atom). The CodeZoo presentation format also makes it very easy to find out how to download the latest release of a project (Figure 5.1 illustrates this for SciPy), although it doesn't say anything

[1]Pentium Pro and upward.

about the deployment method. What we'd like to see in addition to this already great service is a common way for deploying provided software.

Figure 5.1. CodeZoo entry for the SciPy project

In our opinion PyPkg.Installer would make the perfect companion for a CodeZoo type of directory. It should be immensely useful for someone browsing the directory to know that any of its software can be installed through one and the same method, by downloading an installation client (of the PyPkg.Installer type) that takes care of all involved steps. Also, a comprehensive directory of PyPkg-distributed software should only help us in gaining acceptance for our system.

How we should go about putting this into practice is a different question. CodeZoo is already quite expansive, with several hundred different software products, and there's no telling if those responsible for CodeZoo would be interested in our idea. Also, the directory is specially directed at offering components for software development. The PyPkg distribution network is intended to be completely general, and what we would primarily like to see is a Web interface to the PyPkg portal (that presents a directory-like interface outward).

CodeZoo should at least give us some pointers as to how to this can be accomplished. Navigation should be facilitated through some way of grouping portal entries (categories at least), a service for searching projects and their descriptions might also prove useful. Something that we can learn from CodeZoo that is not directly related to presentation is how to cache project descriptions while still keeping them in sync with decentralized representations. That is, the portal could cache project descriptions, to ensure that content is reachable even if a project website should go down, by subscribing to RDF feeds from projects. Such a scheme would, we believe, represent a good compromise between convenience for developers and safety for users.

Appendix A. PyPkg RDF vocabulary

Three RDF vocabularies have been defined using RDFS that together compose the PyPkg RDF vocabulary, they are presented in this appendix in RDF/XML format. The vocabularies `sw` and `pypkg` were introduced as extensions to the `doap` vocabulary, in order to properly support our deployment machinery. The only real purpose of `pypkg` is to define a subclass of `sw:Package` for describing PyPkg packages. `swidx` holds definitions used in creating the PyPkg portal index.

A.1. `sw`

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:doap="http://usefulinc.com/ns/doap#"
>

<!-- Interface -->

<rdfs:Class rdf:about="#Interface">
      <rdfs:comment>A software interface.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="http://usefulinc.com/ns/doap#Project">
   <rdfs:comment>Subclass Interface.</rdfs:comment>
   <rdfs:subClassOf rdf:resource="Interface"/>
</rdfs:Class>

<!-- Implementation -->

<rdfs:Class rdf:about="#Implementation">
   <rdfs:comment>Represent an implementation of a certain interface version, subclass in
order to use revision property of doap:Version.</rdfs:comment>
   <rdfs:subClassOf rdf:resource="http://usefulinc.com/ns/doap#Version"/>
</rdfs:Class>

<rdfs:Property rdf:about="#interface">
   <rdfs:comment>The implemented interface.</rdfs:comment>
   <rdfs:domain rdf:resource="Implementation"/>
   <rdfs:range rdf:resource="Interface"/>
</rdfs:Property>
```

```
<!-- Package -->

<rdfs:Class rdf:about="#Package">
   <rdfs:comment>Generic package representation, subclass in order to indicate package type.
</rdfs:comment>
</rdfs:Class>

<rdfs:Property rdf:about="#size">
   <rdfs:comment>Size of package, in bytes.</rdfs:comment>
   <rdfs:domain rdf:resource="Package"/>
   <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdfs:Property>

<rdfs:Property rdf:about="#download-url">
   <rdfs:comment>URL to package file.</rdfs:comment>
   <rdfs:domain rdf:resource="Package"/>
   <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdfs:Property>

<rdfs:Property rdf:about="#md5">
   <rdfs:comment>MD5 checksum.</rdfs:comment>
   <rdfs:domain rdf:resource="Package"/>
   <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdfs:Property>

<!-- doap:Version -->

<rdfs:Property rdf:about="#package">
   <rdfs:comment>Package for doap:Version.</rdfs:comment>
   <rdfs:domain rdf:resource="http://usefulinc.com/ns/doap#Version"/>
   <rdfs:range rdf:resource="Package"/>
</rdfs:Property>

<rdfs:Property rdf:about="#dependency">
   <rdfs:comment>Declare a dependency of a doap:Version.</rdfs:comment>
   <rdfs:domain rdf:resource="http://usefulinc.com/ns/doap#Version"/>
   <rdfs:range rdf:resource="Implementation"/>
</rdfs:Property>

<rdfs:Property rdf:about="#implements">
   <rdfs:comment>Declare that a doap:Version implements an interface version.
</rdfs:comment>
   <rdfs:domain rdf:resource="http://usefulinc.com/ns/doap#Version"/>
   <rdfs:range rdf:resource="Implementation"/>
</rdfs:Property>
```

```
</rdf:RDF>
```

A.2. `pypkg`

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:sw="http://kangchenjunga.simula.no/arvenk/ns#"
>

<rdfs:Class rdf:about="#Package">
   <rdfs:comment>PyPkg package.</rdfs:comment>
   <rdfs:subClassOf rdf:resource="http://kangchenjunga.simula.no/arvenk/ns/sw#Package"/>
</rdfs:Class>

</rdf:RDF>
```

A.3. `swidx`

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:doap="http://usefulinc.com/ns/doap#"
>

<rdfs:Class rdf:about="#Provider">
   <rdfs:comment>Software provider, associated with doap:Project's. Subclasses doap:Project
in order to use e.g. doap:name</rdfs:comment>
   <rdfs:subClassOf rdf:resource="http://usefulinc.com/ns/doap#Project"/>
</rdfs:Class>

<rdfs:Property rdf:about="#id">
   <rdfs:comment>ID within portal.</rdfs:comment>
   <rdfs:domain rdf:resource="Provider"/>
   <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
</rdfs:Property>

<rdfs:Property rdf:about="#project">
   <rdfs:comment>A project managed by the provider.</rdfs:comment>
   <rdfs:domain rdf:resource="Provider"/>
   <rdfs:range rdf:resource="http://usefulinc.com/ns/doap#Project"/>
</rdfs:Property>

</rdf:RDF>
```

Appendix B. PyPkg Framework Design Document

B.1. Introduction

This document describes the design of the PyPkg agent framework. PyPkg is a generic system for distributing and deploying software, with an initial focus on the Linux platform. The framework is a common base from which PyPkg agents may be created. An agent in this context is a program that operates against the PyPkg distribution network, performing certain tasks on behalf of the user. Such tasks are for the most part related to deployment of software offered through the network; the framework also contains some support for manipulating project representations (in RDF), but this functionality will not be part of the discussion of the design (we choose to concentrate on the deployment functionality). Also, agents are primarily intended as graphical applications, so the framework contains an assortment of PyQt-based GUI components.

The framework model should preferably be abstracted from platform-specifics, so that PyPkg agents may be implemented for other platforms than Linux (Mac OS X and Microsoft Windows come to mind). However, development so far has been conducted on Linux systems only, so significant effort will be required in order to achieve a truly generic model.

When it comes to designing the framework we follow a "demand"-driven process. By this we mean that we decide on a set of demands that the framework is to meet, thus defining the PyPkg model's concept of the deployment problem. The demands may correspond to only part of the real deployment problem; by narrowing the scope (and thereby reducing the complexity) a corresponding solution should be easier to develop, allowing results to be delivered at a speedier rate. The functionality that the PyPkg deployment model is to encompass should be part of this framework, since it makes little sense to assign core functionality to individual agent designs.

First some background on the framework approach, before the set of demands for the framework is defined.

B.1.1. On the framework approach

It was clear from early on that we did not wish to lock the design to a certain type of usage. Personally we've come to enjoy the possibilities presented by full-on package managers, à la Gentoo Portage, where a great deal of deployment functionality is available through the client program, including direct access to distributed software (packages). At the same time we see the need for no-frills "wizard"-style software installers (à la Windows Installer). We've actually concentrated on the latter design, at this stage, in order to support easy installation of a certain software suite, which should serve as proof-of-concept for PyPkg.

In order to facilitate these different scenarios (and possibly some we haven't thought of) core deployment functionality is factored into a highly modular/general framework. Con-

cretely, the framework manifests itself as a hierarchy of Python packages, the top framework package called `pypkg.base`. The `pypkg` toplevel package is intended to hold both clients and framework. The framework package (`pypkg.base`) holds a set of modules implementing general functionality that can be used to assemble clients, there is also a sub-package `pypkg.base.qtgui` which provides core functionality for PyQt based clients.

The framework in its current incarnation is quite well-suited to the needs of different clients, with regard to the goal of accommodating a family of PyPkg clients. Early development was driven by the needs of a comprehensive package manager style client, later in the process focus was shifted toward an "installation script" type of client. The current bias toward the latter means some effort will be required to bring the former up to speed due to changes that has taken place in the framework in the meantime, but the general architecture should be sound.

B.1.2. Demands on the framework

As stated initially in this document, we break the deployment problem into a concrete set of demands that must be satisfied by the PyPkg deployment layer. The framework described in this document is to contain the necessary functionality to make this happen. In order to design a software system with the required functionality, we translate the high-level demands (related to user expectations) into a structured set of responsibilities. That is, we try to deduce a hierarchy of responsibilities that the system must take on in order to deliver expected functionality. The problem is then to turn this formal specification into a functioning system. In order to do so, we choose to consider the system with regard to one such responsibility at a time. In our terminology each responsibility maps to an *aspect* of the system, i.e., we consider only the part of the system that is relevant to the responsibility in question. An aspect doesn't necessarily translate to a single system component.

When designing the framework we therefore operate with specifications on two levels: the set of demands, describing in high-level terms what kind of functionality can be expected, and the hierarchically organized set of aspects that describe how the system relates to individual responsibilities. For each specification we describe each item (demand/aspect) in detail, so to unambiguously express its intent.

The demands

We present now the set of demands on PyPkg deployment functionality that has been defined at this stage. We've defined the deployment functionality to be subject to two main demands: it shall support software *installation* and *uninstallation.* The installation and uninstallation processes are themselves subject to further demands, that serve to detail exactly what users can expect in terms of functionality, for instance that software dependencies are to be resolved as part of the installation process. As such we've grouped the set of demands into two groups, corresponding to the two top-level demands. This is to facilitate a strong understanding of the kind of functionality the framework is to provide.

Support for installation

1 All unsatisfied dependencies shall be satisfied in the installation process

2 Packages in the following formats shall be supported for installation: PyPkg

3 The installation process shall arrange for background acquisition of packages

4 Installation into both system-wide and user-specific locations shall be supported

5 Installations shall be completed by updating the host environment accordingly

The first of these means that when scheduling software for installation, all unsatisfied dependencies toward other software must be met with the help of corresponding packages. The third item relates to the execution of the installation process, rather than its outcome, stemming from the concern that the process be as efficient as possible. If several packages are to be installed in sequence (a dependency chain) the agent might as well save time by downloading queued packages in the background. The fifth item reflects the security model prevalent on Linux systems, where users often lack the rights to install into system-wide locations. The PyPkg policy is to direct installations beneath the user's home directory if he/she lacks administrative rights. Item six pertains particularly to those cases where software is installed outside of system-wide locations (for instance a directory that should be automatically scanned for executables), the typical remedy is to notify of new installation locations through the host environment (most often the shell environment).

Support for uninstallation

1 The uninstallation process shall support removal of orphaned dependencies

2 When uninstalled, an installation's effect on the host environment shall be undone

The first of these items means that installations serving only to satisfy dependencies should be removed when they no longer have any installations depending on them, and are not desirable in themselves, (i.e., they are orphaned). The second item simply states that any environment settings pertaining to an installation shall be cleared when it is uninstalled.

The aspects

As explained earlier, the set of aspects is to be hierarchically organized. The two top-level demands, support for installation and uninstallation, are in our aspectual model translated to the "root" aspects Installation and Uninstallation. Finer-grained demands, related to the top-level ones, are also translated into sub-aspects (descending from Installation and Uninstallation). When naming aspects, we indicate their position in the hierarchy by prefixing their name with the parent's name and a dot ('.'), for instance Installation.DependencyHandling (where DependencyHandling is a sub-aspect of Installation).

The hierarchy of aspects needn't rigidly correspond to the set of demands though. It is used to design a concrete functioning system, and as such one may go into much greater detail for describing how the system functions than what is interesting from the user's point of view (unlike the demands). For instance, in order to acquire packages that are to be installed, there must be some logic for transferring files (packages) over the Internet. Handling network transfers is non-trivial and therefore it makes sense to identify this as a dedicated sub-aspect of Installation. In this version of the design document however, due to time constraints, we'll concentrate on the

"core" aspects (those derived from the specified demands).

- Installation
- Installation.DependencyHandling
- Installation.PackageFormat
- Installation.ParallelDownload
- Installation.Profile
- Installation.Environment
- Uninstallation
- Uninstallation.DependencyHandling
- Uninstallation.Environment

B.1.3. Document structure

The rest of the document contains the actual meat, the detailed description of how the individual aspects of the framework function. To this end, each aspect is assigned a section of its own. The aspects occur in the order they are listed in the previous subsection.

B.2. Installation

The Installation aspect pertains to PyPkg's process for installing software. This process is responsible for integrating a specified project release with the user's system, so that it can be put to use as intended. In order to achieve proper integration, it may not be enough to simply install the release into the filesystem, it may have certain requirements of its own. Requirements that PyPkg must handle as part of the installation process are: dependencies toward other third party software (system components are beyond our scope) and the necessity to export components via the host environment. The latter is normally handled implicitly, by deducing which files are to be publicly available (through PyPkg package structure), but sometimes components are buried in directories that must be explicitly marked for export.

Software installed as part of the installation process must also be registered with the shared PyPkg database, so it may at a later time be inspected and/or managed. Importantly, installations must be registered with sufficient information to allow them to be cleanly removed from the system if the user wishes. PyPkg operates also with an abstract environment stored in the database, that contains settings pertaining to the individual installations. Data in this environment are added to the real host environment, via mechanisms depending on the host platform (on Linux, typically shell variables).

The aspect's design

The most prominent framework component that takes part in the Installation aspect is the class `installation.InstallationManager`. This implements the installation process through

the method `install` and will either install the latest release for a project or a specific project release, in a background thread. The manager exposes a set of signals through which the installation process can be observed, interested objects can connect themselves to signals that are emitted to communicate certain events. For instance, `sigStartingInstall` is emitted to indicate that the installation process has started, while `sigFinishedInstall` indicates that it has successfully ended.

The `install` method of `installation.InstallationManager` must first calculate dependencies for the software to install, that is, a topologically ordered graph of other software that corresponds to dependencies that are not satisfied by the host system. The calculation of the graph is handled by the Installation.DependencyHandling aspect (see Section B.3). When flattened, the graph represents a correctly ordered plan for project releases that must be installed (including the release that was originally scheduled for installation).

With an installation plan in place, corresponding (PyPkg) packages must be acquired before installation can take place. The installation thread queues the packages for download with an object of type `transfer.DownloadManager`, which conducts transfers in a dedicated thread. See Section B.5 for an explanation of this aspect. The installation thread cooperates with the transfer thread through a thread-safe queue. Specifically, the installation manager waits for each entry in its installation plan, in method `_installNode`, for the corresponding package file to be posted via the download queue. Once a package is ready the manager can go on to process it.

First of all, the package is unpacked (as per the Installation.PackageFormat aspect, Section B.4) into a temporary directory. The unpacked content is then registered with a persistent (stored in the database) object representing the installation, of class `installed.InstalledRelease`. Objects of this class will upon instantiation register all files in a directory (containing content from an unpacked package) according to their type (regular file, directory etc.). Such objects also expose logic to install the source files beneath a certain root (through method `install`).

After the object has been instantiated a database transaction is started and the object is inserted into the database, via an "installation catalog" (of class `installed.CompositeCatalog`). The latter exists as a high-level interface for browsing and altering (inserting/removing entries) the database of installations. Only after registering the installation does the manager proceed to integrate the unpacked files with the filesystem (through the `install` method of the `installed.InstalledRelease` object). The Installation.Profile aspect (Section B.6) comes into play here, whether a package is installed in a system-wide or user-specific location depends on the profile (system-wide or user-specific) employed by the PyPkg agent. Once the `installed.InstalledRelease` object has finished installing itself, the transaction is finally committed. Due to enclosing the operations in a transaction a failed installation will not make it into the database proper.

The `installed.InstalledRelease` object also updates environment values in its `install` method. For a description of how this is done see the Installation.Environment aspect, Section B.7. This aspect is also invoked after the whole installation queue has been processed, in order to synchronize itself with the host environment (so that changes during the installation process are reflected in the host environment).

An interesting aspect of recording an installation in the catalog is that it is also registered with its dependencies in the catalog (i.e. it is listed as a dependent of other installations), which makes it possible to count the number of dependency references to an installation.

B.3. Installation.DependencyHandling

This aspect is responsible with turning abstract dependency specifications (toward software interfaces) into topologically ordered dependency graphs, that can be flattened into installation plans. By installation plan we mean a sequence of project releases (`software.Release` instances), that is ordered so that dependencies come before dependents, and where no entry occurs more than once.

A dependency graph is created by instantiating an object of class `installation._InstallGraph` with the release that is to be installed (this will be the root node of the graph). Upon instantiation a recursive topological sorting procedure (`installation._OrderedGraph._recurse`) is invoked, that processes each of a release's dependencies before itself. Each dependency is processed in `installation._InstallGraph._processDependency` which first tries to eliminate the dependency, by checking whether it is satisfied by an installation. Since dependencies are toward interface versions (represented by `software.InterfaceVersion`) this check happens by seeing whether the installation catalog contains an entry that implements a compatible interface version, by invoking the catalog's `getImplementation` method.

If no implementation of the dependency is found among the installations, an implementation must be installed instead. A suitable implementation is found simply by picking the most recent project release (from the software catalog, of class `software.Catalog`) implementing the interface version in question. This release is itself recursively processed, before it is added among the graph's nodes.

Since all of a node's dependencies are added before itself to the graph's list of nodes, the nodes will be correctly sorted once the recursive processing ends. An installation plan can be obtained from a graph through its `queue` attribute, which is just the sequence of project releases corresponding to the graph's nodes.

B.4. Installation.PackageFormat

This aspect pertains to the structure of PyPkg packages and functionality for dealing with them, which comes down to either creating (packing) or unpacking them.

The PyPkg package format consists of a header of metadata followed by a chunk of bzip2-compressed tar archive data. A valid PyPkg package will first contain the identifier "PyPkg v0.1" followed by a line containing an element "cruft" that says how many lines of metadata follows (one per element) before the archive data. At this point, three different metadata elements are defined: "Executable", "Library" and "Python". These metadata elements are used to advertise directories containing executables ("Executable"), object libraries ("Library") or Python modules ("Python"), that should be published through the environment (directories with executables are added to the `PATH` variable for Linux shells for instance).

Package content (archived files) is organized according to a peculiar scheme. The upper level of the contained directory structure must consist of "symbolic directories", some of which are: "@Executable", "@Library", "@PythonSite". The full list of symbolic directories is exposed as `packageformat.symbolicDirectories`. Packaged files should then be placed beneath symbolic directories according to their type: executables go into "@Executable", object libraries go into "@Library", Python modules go into "@PythonSite" etc.

The reason for this scheme is twofold. Linux filesystem layouts typically organize files according to their type (executables are installed into /usr/local/bin etc.), our symbolic directories can be translated into the host system's layout. Also, many types of files are intended to be found by certain applications (for instance Python modules should be known to the Python interpreter), so if they are installed outside of standard locations their location is instead advertised through some environment mechanism (Python can use the shell variable `PYTHONPATH` for extending its module search path for instance).

Logic pertaining to PyPkg packages, packing and unpacking, are contained in the package (in the UML sense) `packageformat` which exposes functions `pack` and `unpack`. Before `pack`'ing a directory, its content must be translated according to our symbolic directory scheme. `unpack` will simply unpack a package into the destination directory, not translating the symbolic directories. Instead, the `_filesystemscheme.translateDirectoryContents` function is provided for this purpose.

B.5. Installation.ParallelDownload

This aspect exists in order to allow PyPkg packages to be acquired in a background thread, so that already acquired packages can be installed at the same time, thus eliminating overhead where the installation manager must constantly wait for a package to be downloaded before it can process it.

The aspect presents itself through `transfer.ThreadedDownloadManager`. An object of this class lets other objects schedule a sequence of Web resources (represented by URLs) to be downloaded, through the method `__call__`, in a dedicated thread. The `__call__` method also takes an argument `slot` that is called for each resource that has finished downloading, with the path to the corresponding file. `installation.InstallationManager` utilizes this argument so that file paths are pushed on a queue for inter-thread communication.

The actual downloading of Web resources happens with the help of the PycURL library, which presents a higher-level abstraction of the traditional socket mechanism. Specifically, it presents an asynchronous "transfer stack". The stack constantly operates with a number of transfers that are processed chunk-wise. That is, PycURL will process as much data as there are available, then we wait for more incoming data by way of e.g. `select.select` (in the standard Python library). This continues until the stack is exhausted (downloads are finished).

B.6. Installation.Profile

In order to support deployment in both system-wide and user-specific mode we've introduced the notion of *profiles*. An agent operates according to a certain profile, a global configuration which can affect its operation in a number of ways. Most importantly it determines whether software is installed on a per-user basis or (the user-specific profile) available to all users in the system (the system-wide profile).

Since the profile in use governs the global configuration of an agent it is represented by a global object, `common.core`, of class `core._Core`. This object contains state that is affected by the choice of profile, notable attributes (of `common.core`) include: `conf.destinationRoot` and `database.conf.destinationRoot` governs where to direct installations, for the system-

wide profile it will equal the system root, for the user-wide profile it will be ~/.pypkg/root. `database`, which can either be the user or system database, is the database that should be modified when registering installations and updating environment settings. This means that when conducting an installation, it isn't necessary to check which database should be manipulated (the user or system database), it is decided already by the `common.core` object.

B.7. Installation.Environment

This aspect concerns itself with manipulating host environment settings in conjunction with installation of software. At the moment such manipulation is necessary in order to publish locations of certain types of files, the archetypal example being executables. If an installation involves a directory of executables that is outside the standard path, and these executables should indeed be in the path, the path must be adjusted (on Linux, by augmenting the `PATH` shell variable) correspondingly. Other examples of files that may need to be published (or "exported") via the environment include object libraries and Python modules.

We've approached this problem by mapping installations to environment settings in the database, a functionality which is handled by class `_environment.Environment`. Concretely, this class exposes a method `registerInstallation` which will register directories belonging to an installation depending on their content. That is, the installation may for instance have associated with it a directory "python-site" that contains Python modules. This directory is added to a database index of directories that should be added to the Python interpreter's module path.

The internal environment representation is a nice and consistent abstraction, but it must be translated into host environment settings in order to be of any use. The `Environment.update` method serves to synchronize the abstract PyPkg environment with the host environment. On Linux we currently employ two different mechanisms for manipulating the host environment, shell variables for the general case and a special `sitecustomize` module for setting up the Python interpreter (the location of our sitecustomize.py file must be placed in the interpreter's path via the `PYTHONPATH` variable though).

The shell environment is set up by augmenting path variables corresponding to types of exported files: executables -> `PATH`, shared object libraries -> `LD_LIBRARY_PATH`, manpages -> `MANPATH`, sitecustomize.py -> `PYTHONPATH` (other Python modules are handled in sitecustomize.py). We employ a layered approach to manipulating shell variables, in order to support different shells and to update paths in an intelligent manner. A line is added to the startup files of supported shells (currently bash and zsh) which evaluates ("sources") a portable shell file belonging to PyPkg, which in turn invokes a special Python script (generated by the `update` method) to generate updated path variables.

When it comes to Python modules we've chosen to go with the `sitecustomize` option for modifying the module search path, over `PYTHONPATH`, since it gives us a whole lot more flexibility. What our special `sitecustomize` does is first execute any other `sitecustomize` in the interpreter's path (so as to respect any user preferences) before adding directories belonging to PyPkg installations to the front of the module path. One possibility of this approach that we haven't exploited yet is to add installations to the module path only if they are compatible with the interpreter version.

B.8. Uninstallation

This aspect pertains to PyPkg's uninstallation process. The uninstallation process is essentially the inverse of the installation process (Section B.2), in that it serves to undo the effect of a given installation. As such, its responsibility is to remove the installation's files from the filesystem, clear any associated settings in the host environment and also uninstall any dependencies that are orphaned in the process (an orphaned installation is no longer depended on and not marked as desired in itself).

The uninstallation process operates on data that is registered in the shared PyPkg database by the installation process, and is entirely reliant on entries in this database for conducting uninstallations. That is to say, it needs an exact representation of how an installation has affected the host system in order to remove it. Thus, the installation process must for each installation register its associated files, its environment settings and how it relates to other installations (through references to dependencies and dependents). The uninstallation process is of course responsible for removing all traces of an installation from the database as part of it uninstallation.

The aspect's design

The public interface to the Uninstallation aspect is the same as for the Installation aspect actually, class `installation.InstallationManager`. It might seem like somewhat of a paradox that the installation manager should be responsible for the uninstallation process, but it is a good way to ensure synchronized access to the installation database. It might've been better design to define separate managers and ensure synchronization in a lower, common, layer, but the current design is at least functional (and the merit of separate managers is arguable).

Mirroring the Installation aspect, `installation.InstallationManager` provides method `uninstall` to initiate the uninstallation process for a certain installed release. This will generate an uninstallation task that is scheduled with the same background thread as installation tasks (thereby synchronizing access to the installation database). This process can also be observed through a set of signals, for instance, `sigStartingUninstall` is emitted when the uninstallation process is starting and `sigFinishedUninstall` when it has ended successfully.

The handling of dependency relations during the uninstallation process differs from that during installation. When a release is first considered for uninstallation, the manager checks whether there are any other installations that depend on it. If so, it will refuse to continue processing by raising `installation.DependedOnError`. Otherwise dependent installations would likely break. The next step is to calculate an uninstallation plan including orphaned dependencies, meaning installed releases which number of dependents drops to zero. This is a type of garbage collection, so that installations no longer hang around when their original purpose (satisfying dependencies) is gone.

Similar to how an installation plan is generated, a topologically ordered uninstallation graph is created by the Uninstallation.DependencyHandling aspect (Section B.9) and flattened (into a serialized plan).

In comparison to the Installation aspect, the processing of individual entries in the plan is quite simple. There's no need to download anything, so each entry is simply passed in sequence to method `installation.InstallationManager._uninstallNode`. This method starts a database transaction before it removes the release, which is an instance of `in-`

`stalled.InstalledRelease`, from the installation catalog and tells it to uninstall itself through its method `uninstall`. The transaction is then committed.

Again mirroring the approach of the Installation aspect, the `uninstall` method of `installed.InstalledRelease` takes care of clearing its associated settings from the PyPkg environment representation (as per aspect `Uninstallation.Environment`, Section B.10). The PyPkg environment is also synchronized with the host environment after the whole uninstallation plan has been processed.

B.9. Uninstallation.DependencyHandling

This aspect is responsible with calculating a topologically ordered graph of unneeded dependencies based on a release that is to be uninstalled, this graph can be flattened into a serialized uninstallation plan. The background is that when a project release is installed it may refer to a number of other installed releases as dependencies, that is, they satisfy dependencies of the release in question. When the release is to be uninstalled it is no longer registered as a dependent with its dependencies, those of its dependencies that as a result have no dependents (reference count is zero) are considered orphans. Unless an orphan is marked as desirable on its own (releases that are explicitly installed by the user are marked this way) it is scheduled for uninstallation by adding it to the graph. This process is followed recursively for each node that is added to the graph, so that their dependencies are considered as well.

A graph of releases to be uninstalled is created by instantiating an object of class `installation._UninstallGraph` with the release that is to be uninstalled, which becomes the root node of the graph. Similar to how it is done for `installation._InstallGraph` (refer to Section B.3), a recursive topological sorting procedure is invoked upon instantiation that processes each of a release's dependencies before itself. The two graph classes are both based on `installation._OrderedGraph`, so they have a lot in common. There is a difference, though, in that before a release's dependencies are recursively processed (in `installation._OrderedGraph._recurse`) the release is unregistered with each dependency (in method `installed._UninstallationGraph._aboutToProcess`). Dependencies that are still being depended on, by other releases, are ignored (not eligible for uninstallation).

Dependency relations are indicated by associating each installed release, an `installed.InstalledRelease` object, with a number of *dependencies* and *dependents*. Each such object has a `dependencies` attribute and a `dependents` attribute, the former contains references to other releases it depends on, while the latter contains references to releases that depends on it. So, when a release is added to the uninstallation graph it is removed from the `dependents` set of each of its dependencies (listed in its `dependencies` attribute).

Since the topologically sorted graph will come out dependencies before dependents (which is how we want it in the installation case), the last step is to reverse its nodes. If the uninstallation process should for some reason finish prematurely it is better that dependents are removed first.

B.10. Uninstallation.Environment

This aspect concerns itself with manipulating host environment settings in conjunction with uninstallation of software. It represents a reversal of changes made as part of the Installa-

tion.Environment aspect (Section B.7), in that any effect of an installation on the host environment is undone.

This aspect is also handled by class `_environment.Environment`, through the method `unregisterInstallation` which takes an `installed.InstalledRelease` instance as argument. Clearing an installation's settings from the environment is quite easy since the environment maps settings to installations (via database indexes). Consequently, `unregisterInstallation` only needs to look up the installation in its indexes and remove corresponding settings.

The host environment is updated to reflect modified settings in the `update` method of `_environment.Environment`, which simply translates the current state of the abstract PyPkg environment into host environment settings. Refer to Section B.7 for an explanation of how this works.

References

[1] The AspectJ Team. The AspectJ Programming Guide, Palo Alto Research Center, 2003. URL *http://eclipse.org/aspectj/doc/released/progguide/index.html*.

[2] Automatically Tuned Linear Algebra Software (ATLAS), 30 January 2006. URL *http://math-atlas.sourceforge.net/faq.html*.

[3] Edward C. Bailey. *Maximum RPM - Taking the Red Hat Package Manager to the Limit*. Red Hat Software, Inc., February 1997. URL *http://www.rpm.org/local/maximum-rpm.ps.gz*.

[4] Dave Beckett, Eric Miller and Dan Brickley. Expressing Simple Dublin Core in RDF/XML, 31 July 2002. URL *http://dublincore.org/documents/dcmes-xml/*.

[5] Dave Beckett. Turtle - Terse RDF Triple Language, 20 December 2005. URL *http://www.dajobe.org/2004/01/turtle/*.

[6] Tim Berners-Lee. Semantic Web Roadmap, September 1998. URL *http://www.w3.org/DesignIssues/Semantic.html*.

[7] Tim Berners-Lee. Cool URIs don't change, 1998. URL *http://www.w3.org/Provider/Style/URI.html*.

[8] Internet Protocols (IP), Cisco Systems, Inc., 14 December 2005. URL *http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ip.htm*.

[9] Wolfram Conen and Reinhold Klapsing. A Logical Interpretation of RDF. *Linkoping Electronic Articles in Computer and Information Science* **5** (26 August 2000). URL *http://citeseer.ist.psu.edu/conen00logical.html*.

[10] conduit, Dictionary.com, 27 December 2005. URL *http://dictionary.reference.com/search?q=conduit*.

[11] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, pages 60-66. Springer-Verlag, 1982.

[12] Fred L. Drake. distutils – Building and installing Python modules, 28 September 2005. URL *http://docs.python.org/lib/module-distutils.html*.

[13] Edd Dumbill. Finding Friends with XML and RDF. The Friend-of-a-Friend vocabulary can make it easier to manage online communities, 1 June 2002. URL *http://www-128.ibm.com/developerworks/xml/library/x-foaf.html*.

[14] Edd Dumbill. Uniquely identifying projects, 24 March 2004. URL *http://www-128.ibm.com/developerworks/xml/library/x-osproj2/#h2*.

[15] Edd Dumbill. XML Watch: Describe open source projects with XML, Part 1. Keep project information up-to-date with the DOAP vocabulary, 26 February 2004. URL *http://www-128.ibm.com/developerworks/xml/library/x-osproj.html*.

[16] Edd Dumbill. XML Watch: Describe open source projects with XML, Part 3. A first draft of the DOAP vocabulary, 11 June 2004. URL *http://www-128.ibm.com/developerworks/xml/library/x-osproj3/*.

[17] Python Enthought Edition, Enthought, Inc., 30 January 2006. URL *http://code.enthought.com/enthon/*.

[18] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, Mehmet Aksit. *Aspect Oriented Software Development*. Addison Wesley, October 2004.

[19] Free Software Foundation. GNU General Public License, June 1991. URL *http://www.gnu.org/copyleft/gpl.html*.

[20] Free Software Foundation. GNU Make, Free Software Foundation Inc., 25 December 2002. URL *http://www.gnu.org/software/make/manual/make.html*.

[21] The Free Software Definition, 12 February 2005. URL *http://www.fsf.org/licensing/essays/free-sw.html*.

[22] Achim Gottinger, Mark Guertin, Nicholas Jones, Mike Frysinger. ebuild - the internal format, variables, and functions in an ebuild script, Gentoo Foundation, 30 April 2005. URL *http://gentoo-wiki.com/MAN_ebuild_5*.

[23] Mike Hearns. Autopackage - Introduction to the Next Generation Linux Packaging, 2002. URL *http://www.osnews.com/story.php?news_id=2307&page=1*.

[24] Eric Jones, Travis Oliphant, Pearu Peterson and others. SciPy: Open Source Scientific Tools for Python, 2001 -. URL *http://www.scipy.org*.

[25] Graham Klyne, Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, 10 February 2004. URL *http://www.w3.org/TR/rdf-concepts/*.

[26] The Injector: Design, 27 September 2005. URL *http://zero-install.sourceforge.net/injector-design.html*.

[27] David MacKenzie, Ben Elliston, Akim Demaille. *Autoconf*. Creating Automatic Configuration Scripts. Free Software Foundation Inc., November 2003.

[28] Windows 2000 Professional: Windows Installer Service, 5 December 2005. URL *http://www.microsoft.com/technet/prodtechnol/windows2000pro/evaluate/featfunc/winspro.mspx*.

[29] Eric Miller. Semantic Web Activity Statement, 2005. URL *http://www.w3.org/2001/sw/Activity*.

[30] What's This? DOAP-over-Atom, CodeZoo, 17 January 2005. URL *http://*

www.codezoo.com/about/doap_over_atom.csp.

[31] What's This? DOAP, CodeZoo, 20 December 2005. URL *http://ruby.codezoo.com/about/doap.csp.*

[32] Information regarding SourceForge.net-provided RSS feeds, Open Source Technology Group, 18 January 2006. URL *https://sourceforge.net/docman/display_doc.php?docid=15483&group_id=1.*

[33] freshmeat, Open Source Technology Group, 30 January 2006. URL *http://freshmeat.net*.

[34] SourceForge.net, Open Source Technology Group, 30 January 2006. URL *http://www.sourceforge.net*.

[35] Pearu Peterson. F2PY, 30 January 2006. URL *http://cens.ioc.ee/projects/f2py2e/*.

[36] What is Python?, Python Software Foundation, 30 January 2006. URL *http://www.python.org/doc/Summary.html.*

[37] Psyco, Armin Rigo, 30 January 2006. URL *http://psyco.sourceforge.net*.

[38] Rene Rivera, David Abrahams, Vladimir Prus. Boost.Jam, 18 November 2004. URL *http://www.boost.org/tools/build/jam_src/index.html*.

[39] PyQt, Riverbank, 30 January 2006. URL *http://riverbankcomputing.co.uk/pyqt*.

[40] Stuart Russell, Peter Norvig. *Artificial Intelligence - A Modern Approach.* Prentice Hall. Second Edition, 2003.

[41] Building ATLAS for SciPy, 14 December 2005. URL *http://www.scipy.org/documentation/buildatlas4scipy.txt*.

[42] Numeric, 30 January 2006. URL *http://numeric.scipy.org/*.

[43] Simula Research Laboratory. Scientific Computing, 30 January 2006. URL *http://www.simula.no/departments/scientific/*.

[44] Vermeulen et al. A Portage Introduction, 2005. URL *http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1.*

[45] W3C. World Wide Web Consortium Issues RDF and OWL Recommendations. Semantic Web emerges as commercial-grade infrastructure for sharing data on the Web, 10 February 2004. URL *http://www.w3.org/2004/01/sws-pressrelease.*

[46] W3C. RDF Vocabulary Description Language 1.0: RDF Schema, 10 February 2004. URL *http://www.w3.org/TR/rdf-schema/*.

[47] W3C. RDF/XML Syntax Specification (Revised), 10 February 2004. URL *http://www.w3.org/TR/rdf-syntax-grammar/*.

[48] Advanced Packaging Tool, 10 August 2005. URL *http://en.wikipedia.org/wiki/Ad-*

vanced_Packaging_Tool.

[49] Deb (file format), Wikipedia, 1 December 2005. URL *http://en.wikipedia.org/wiki/.deb*.

[50] MD5, Wikipedia, 10 December 2005. URL *http://en.wikipedia.org/wiki/MD5*.

[51] Object-oriented programming, Wikipedia, 17 December 2005. URL *http://en.wikipedia.org/wiki/object-oriented*.

[52] Windows Update, Wikipedia, 20 August 2005. URL *http://en.wikipedia.org/wiki/Windows_Update*.

[53] Pentium Pro, Wikipedia, 25 January 2006. URL *http://en.wikipedia.org/wiki/Pentium_pro*.

Glossary

API *page 4*

Application Programmers Interface: a collection of programming language constructs to aid in writing programs.

AGENT *page 17*

A concept rooted in the AI tradition, basically an autonomous entity that is capable of performing tasks on behalf of a user. Typically implemented by a computer program.

ASPECT *page 17*

The appearance of an object from a specific vantage point.

BINARY RELATION *page 20*

A relation between two entities.

BOOTSTRAPPING *page 46*

Preparing for execution.

CONDUIT *page 12*

A channel for transferring fluids (in the abstract sense: data).

DEPLOYMENT *page 1*

The act of installing software on target systems, and related tasks such as updating or removing installations.

FRAMEWORK *page 6*

In software development, a reusable skeleton for basing concrete applications on.

GUI *page 6*

Graphical User Interface.

INSTALLATION *page 1*

A piece of software installed in a computer system.

JIT COMPILER *page 6*

A Just-In-Time compiler translates bytecode (pseudo machine code) into native machine code at runtime, to strike a balance between portability and performance.

LINUX DISTRIBUTION *page 1*

A pre-packaged composition of the Linux kernel and discrete components that together form an operating system.

OBJECT LIBRARY *page 32*

A binary code library, on Linux normally in the ELF format.

PACKAGE MANAGEMENT *page 4*

A way of managing the installation and subsequent management (updating, uninstallation) of software (in the form of “packages”), very much a Unix (Linux in particular) paradigm.

PATH *page 4*

A list of filesystem directories, to be traversed for some purpose. The perhaps best known example being the shell path, which is used to locate executable programs.

PLATFORM-AGNOSTIC *page 1*

Not committed to the specifics of any particular platform.

PYTHON PACKAGE *page 6*

A collection of Python modules/packages.

RSS (RICH SITE SUMMARY) *page 11*

A format for conveying news items from websites to subscribers.

SEMANTIC ASSERTION *page 18*

A commitment to a certain fact, typically used to explicitly state properties of objects.

SEMANTICS *page 20*

Meaning in language.

SHARED OBJECT LIBRARY *page 32*

An object library that can be dynamically loaded into programs at load- or runtime, corresponding to Windows .dll files and Linux .so files.

SHELL *page 5*

A textual user interface to the operating system, for example the ubiquitous Unix Bourne shell.

SHELL SCRIPT *page 4*

A scripting language interpreted by Bourne-compatible Unix shells.

SOFTWARE DIRECTORY *page 11*

A website presenting a catalog of software, organized in terms of projects.

SOFTWARE PACKAGE *page 1*

A piece of software packaged according to a certain format, with deployment in mind.

THREAD *page 32*

A program may process several simultaneous tasks by splitting execution into so-called threads.

WIZARD *page 10*

A in interactive program for leading the user through a complex task, via a step-by-step dialog.

XML *page 1*

Extensible Markup Language: a general-purpose SGML-derived markup language for data description.